Entwicklung von Environment-Shadern in Cg und CgFX unter Berücksichtigung des Workflows zur Erstellung virtueller Szenenbilder im WDR Köln

Bachelorarbeit
im Institut für Medien- und Phototechnik
an der Fachhochschule Köln

Autor
Heike Dahlbüdding
Mat.-Nr.: 11062102

Referent: Prof. Dr. Stefan M. Grünvogel
Korreferent: Ing. T. Behle, Westdeutscher Rundfunk Köln

Köln, im Februar 2011

Development of environment shaders in Cg and CgFX considering the workflow of creating virtual production design at the WDR Cologne

Thesis
at the Institute of Media and Phototechnology
University for Applied Science Cologne

Author
Heike Dahlbüdding
Mat.-Nr.: 11062102

First Reviewer: Prof. Dr. Stefan M. Grünvogel
Second Reviewer: Ing. T. Behle, Westdeutscher Rundfunk Köln

Cologne, February 2011

Kurzbeschreibung

Titel:	Entwicklung von Environment-Shadern in Cg und CgFX unter Berücksichtigung des Workflows zur Erstellung virtueller Szenenbilder im WDR Köln
Autorin:	Heike Dahlbüdding
Referenten:	Prof. Dr. Stefan M. Grünvogel, FH Köln Dipl. Ing. Torsten Behle, WDR Köln
Zusammenfassung:	In der vorliegenden Arbeit werden zwei Environment-Shader entwickelt, einer für die Software Maya, der zweite für die im WDR Köln genutzte Grafiksoftware 3Designer. Zur Entwicklung wird der Worflow zur Erstellung virtueller Szenenbilder im WDR Köln berücksichtigt. Dafür wird der Workflow vorgestellt und die Umsetzung in den Shader detailliert erläutert. Als Ergebnis liegt für jedes Programm ein Shader vor, der bewirkt, dass die Reflexion einer Umgebung, die als Kugel vorliegt, auf dem jeweiligen Objekt erkennbar ist. .
Stichwörter:	Shader, Cg, CgFX, Environment-Mapping, WDR Köln
Datum:	24.02.2011

Abstract

Title:	Development of environment shaders in Cg and CgFX considering the workflow of creating virtual production design at the WDR Cologne
Author:	Heike Dahlbüdding
Reviewers:	Prof. Dr. Stefan M. Grünvogel, FH Köln Dipl. Ing. Torsten Behle, WDR Köln
Abstract:	In this bachelor thesis two environment shaders are developed, one for the software Maya, the other for 3Designer, a graphical software used by WDR Cologne. For this development the workflow of building virtualproduction design at WDR Cologne is taken into account. Therefor the workflow is introduced and the implementation of the shader is explained. As a result there is one shader for each program which causes, that the reflection of a sphere environment is visible on the particular object.
Key Words:	Shader, Cg, CgFX, Environment-Mapping, WDR Köln
Date:	24.02.2011

Inhaltsverzeichnis Seite

1 Einleitung und Problemstellung

Der Westdeutsche Rundfunk (WDR) für das Land Nordrhein Westfalen, ist seit Dezember 1965 eines der größten Rundfunkunternehmen Europas mit dem Hauptsitz in Köln.

Seit 1997 wird im Studio E des WDR Köln virtuell produziert. Der Moderator, bzw. die Moderatorin befinden sich im Studio in einem grünen Raum, in dem als Teil des realen Sets selten mehr als ein Tisch oder Sitzgelegenheiten stehen. Der grüne Anteil des Bildes, den die Kamera aufnimmt, wird für das Fernsehbild entfernt (gestanzt) und ein virtuelles Studio anstelle des Grüns gezeigt.

Das Szenenbild des gesamten Studios wird am Computer erstellt. Die Vorteile dieser Produktionsart bestehen darin, dass eine kurze Zeit für die Inbetriebnahme eines Sets benötigt wird. Zeitaufwändige Auf- und Abbauten der Kulissen sind nicht mehr notwendig und es kann so schnell zwischen den Sets gewechselt werden.

Durch Sensoren an den Kameras ist es möglich ihre Bewegung zu ermitteln und diese in das virtuelle Szenenbild zu übertragen. So entsteht der Eindruck, dass sich die Kamera durch das virtuelle Studio bewegt. Die Bilder, die gerendert (Umwandlung der Szene in ein zweidimensionales Bild) werden, müssen so schnell hintereinander erstellt werden, dass die Bildfolge als dynamischen Prozess empfunden wird. Die Studiokamera nimmt die Moderatorin oder den Moderator mit 50 Bildern pro Sekunde auf. Diese Zeit steht damit den Rechnern zur Verfügung. Echtzeit spielt daher eine große Rolle. Je mehr in einem Szenenbild zur Erstellung eines Bildes berechnet werden muss, desto länger dauert es. Möchte man z.B. Reflexionen physikalisch genau berechnen, so ist die Produktion in Echtzeit nicht möglich, da dies einen zu großen Rechenaufwand darstellt. Doch ohne Reflexionen wirkt ein Szenenbild starr und leblos. Denn sobald sich eine Kamera im Studio bewegt, müssten sich auf spiegelnden und glänzenden Oberflächen vorhandene Reflexionen verändern. Die Einführung von programmierbaren Shader-Einheiten auf modernen Grafikkarten ermöglicht es, solche Effekte in Echtzeit zu erzielen. Die darauf laufenden Shaderprogramme berechnen Reflexionen zwar nicht physikalisch genau, aber

für den Betrachter entsteht der Eindruck einer reflektierenden Oberfläche. Das Szenenbild wirkt damit lebendiger und realistischer.

Im virtuellen Studio des WDR Köln wird Soft- und Hardware der Firma Orad eingesetzt. Zusätzlich nutzen die Szenenbildner Maya, von der Firma Autodesk, zum Design (Modellierung und Lichtberechnung).

Das Ziel dieser Arbeit ist es für Maya und 3Designer (der Firma Orad) jeweils einen Environment-Shader zu entwickeln, der die Reflexion einer beliebig wählbaren Umgebung darstellt, welche mittels spezieller Techniken von beliebigen Orten aufgenommen werden kann. Diese Shader sollen optimal an den Workflow zur Erstellung der Szenenbilder angepasst werden.

Für die Programmierung in der Software 3Designer steht die High-Level-Shading-Sprache Cg (C for grafics), für die Implementierung in Maya CgFX (Cg für Effekte) zur Verfügung.

2 Grundlagen

2.1 Einführung

2.1.1 Grundbegriffe

In der Computergrafik werden in einer dreidimensionalen Szene Objekte modelliert, die später in Bilder umgerechnet werden, bei denen die Beleuchtung der Szene, Materialien des Objektes und Schattierungen mit berücksichtigt werden. Die Umwandlung der dreidimensionalen Szenen in ein zweidimensionales Bild, wird Rendern genannt.
Die Objekte bestehen aus grafischen Primitiven, zu denen Punkte, Linien und Polygone (bevorzugt Dreiecke) gehören. Die einzelnen Polygone bestehen aus Vertices und Fragmenten. Die Eckpunkte der Polygone heißen Vertices. Zwischen den Eckpunkten liegen die Fragmente. Sie speichern Informationen (wie z.B. Farbe, Koordinate oder den Alphawert) und dienen dazu die Pixel einzufärben. Pixel sind das Ergebnis eines Pipeline-Prozesses (in Kapitel 2.5. dargestellt) und werden z.B. auf einem Bildschirm dargestellt. Sie besitzen nur noch eine Farbe, wobei das Fragment die Informationen zur Bestimmung der Farbe beinhaltet. [1]

2.1.2 Rendering

Es gibt verschiedene Arten ein Bild zu rendern. Früher wurde ausschließlich offline gerendert, das bedeutet, dass der Renderprozess auf der CPU (CPU bedeutet central processing unit (= Hauptprozessor) und ist die zentrale Verarbeitungseinheit eines Computers) stattfand und die Berechnung der Bilder sehr lange dauerte. Es war damit aber ein hochqualitativer Renderprozess möglich. Dieses Verfahren wird auch Software-Rendering genannt. Verfahren, wie das Ray Tracing[1] laufen immer noch offline ab.
Mit der Verlagerung des Rendering-Prozesses auf die GPU (GPU bedeutet graphics prozessing unit (= Grafikprozessor) und ist ein Teil der Grafikkarte) wurde Echtzeit ermöglicht. Das Hardware- oder online-Rendering entlastete die CPU und die Prozessorzeit konnte für andere Aufgaben genutzt werden. Im Vergleich zum offline-Rendering waren die Ergebnisse qualitativ schlechter, da aus Geschwindigkeitsgründen

[1] Beim ray tracing wird der Strahl von der Kamera in die Szene berechnet. Es können Spiegelungen und Reflexionen innerhalb der Szene berechnet werden und Objekte, die von der Kamera aus von anderen verdeckt werden nicht mit in die Bildberechnung mit eingehen.

vereinfachte Verfahren benutzt werden mussten. Mit Einführung der Shader kann mittlerweile auch in Echtzeit ein gutes Ergebnis im Hardware-Rendering erzielt werden. [2]

2.1.3 OpenGL und Direct3D

OpenGL und Direct3D sind konkurrierende Standards für die Programmierung dreidimensionaler Szenen. Sie definieren APIs (Application Programming Interfaces), welche die Grafikprogrammierung von der Grafikhardware und deren Treibern abstrahiert. Durch sie ist es möglich, Programme mit 3D-Darstellung ohne Code-Änderungen auf verschiedensten Hardware-Plattformen und Betriebssystemen auszuführen.
OpenGL (Open Graphics Library) wird als offener Standard entwickelt und ist betriebssystem- und programmiersprachenunabhängig. Direct3D ist ein Bestandteil von DirectX, welches neben Direct3D noch andere Schnittstellen beinhaltet um Hardware anzusprechen, wie z.B. DirectSound. DirectX ist alleiniges Eigentum von Microsoft. Damit unterstützt es lediglich die Windows-Plattform. [3, Seite 4f]

2.2 Beleuchtungsmodelle

Beleuchtung und Schattierung von dreidimensionalen Objekten sind wesentliche Elemente, die einen dreidimensionalen Eindruck beim menschlichen Beobachter hervorrufen. Wenn alle Polygone die gleiche Farbe haben, existieren keine Abschattungen und das Objekt erscheint flach. Erst durch das Rendern mit Beleuchtung und Schattierungen wirkt ein Objekt dreidimensional.
Beleuchtungsmodelle bestimmen die Farbe eines beleuchteten Vertex auf der Oberfläche eines Objektes. Möchte man in der Computergrafik beleuchtete Objekte darstellen, so ist dies immer nur eine Näherung an die korrekte physikalische Lichtberechnung, wozu die Transmission von Licht, Verdeckungen und Reflexionen gehören. Sehr nahe kommen zwar globale Beleuchtungssysteme, bei denen sowohl direkte Lichtquellen als auch indirekte, die durch Reflexionen der Lichter von Oberflächen andere Objekte entstehen, berücksichtigt werden. Nutzt man solche Verfahren, bedeutet das allerdings einen hohen Aufwand und damit eine lange Rechenzeit. Hierzu gehören u.a. das Ray Tracing- und Radiosity-Verfahren.

Vereinfacht wird die Beleuchtung durch lokale Beleuchtungsmodelle dargestellt, wie bei den Modellen von Lambert und Phong. Diese berücksichtigen nicht die Objekte in der Umgebung und deren Reflexionen, sondern nur das direkte Licht der Lichtquellen. Für alle anderen Lichttypen wird pauschal ein ambienter Anteil zum Lichteinfall hinzu addiert. Diese Verfahren werden in Echtzeit genutzt.

2.2.1 Lambert-Beleuchtungsmodell

Dieses Modell simuliert die Beleuchtung von ideal diffus reflektierenden Oberflächen. Ideal diffus bedeutet, dass der reflektierende Anteil des Lichtes auf einen Punkt einer Oberfläche in alle Richtungen gleich ist. So spielt die Orientierung des Oberflächenstücks gegenüber der Betrachtungsrichtung keine Rolle. Allerdings besteht eine Abhängigkeit vom Winkel zwischen dem einfallenden Lichtstrahl und der Oberflächennormalen zu der Intensität des diffus reflektierten Lichtes. Denn nur der effektive Flächeninhalt einer Oberfläche, den eine Lichtquelle auch bestrahlen kann, wird von dieser mit voller Intensität beleuchtet. Dies wird im Lambertschen Gesetz beschrieben (vgl. Abbildung 1, wobei A das Flächenstück des Objektes, $A_\perp$ das effektiv beleuchtete Flächenstück, n die Flächennormale und l die Lichtrichtung darstellen.):

$$|A_\perp| = |A| \cos\theta_1$$ **(Formel 1: Lambertsches Gesetz)**

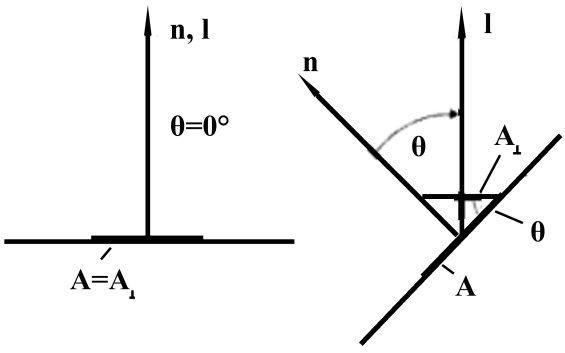


Abbildung 1: Lambertsches Gesetz [3, Seite 280]

Es kann also nur die Helligkeit reflektiert werden, die auch auf der Fläche ankommt.

Die reflektierte Intensität ist ebenfalls proportional zu $|A_{\perp}|$ und cos θ. Die Formel des Lambert-Beleuchtungsterm ergibt sich zu:

$$I_d = I_l r_d \max(0, \langle l, n \rangle)$$ **(Formel 2: Lambert Beleuchtungsterm)**

Wobei I_d die Intensität des diffus reflektierten Lichtes, I_l die Intensität des von der Lichtquelle einfallenden Lichtes und r_d den diffuse Reflexionskoeffizient darstellen. Mittels r_d lässt sich einstellen, welcher Anteil des einfallenden Lichtes diffus gestreut wird. Da es nur um die Vorderseiten der Oberflächen geht, wobei θ zwischen -90° und 90° liegt, wird mit dem Maximum gerechnet. Der Cosinus des Winkels θ wird mittels des Skalarproduktes zwischen l und n dargestellt: cos $\theta = \langle l, n \rangle$. Abbildung 2 verdeutlicht die Abhängigkeit der Intensität des reflektierten Lichts vom Einfallswinkel des eintreffenden Lichtes.

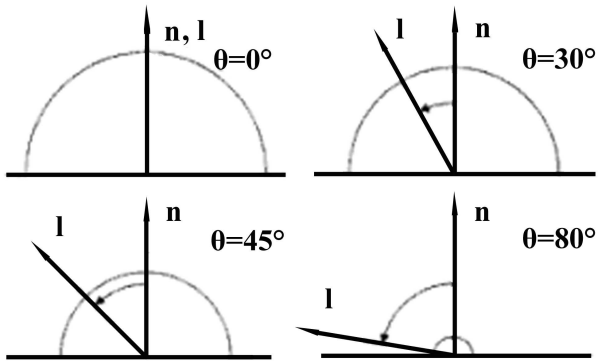


Abbildung 2: Abhängigkeit der Intensität vom Blickwinkel [3, Seite 280]

2.2.2 Phong-Beleuchtungsmodell

Das Phong-Beleuchtungsmodell erweitert das Lambert-Beleuchtungsmodell um den Phong Beleuchtungsterm, der eine unvollkommene Reflexion modelliert und somit Glanzlichter auf den Oberflächen der Objekte mit einbezieht. Unvollkommene Reflexion bedeutet, dass im Gegensatz zu einer ideal spiegelnden Reflexion, bei der das Licht in genau die Richtung gespiegelt wird, die man mit dem Reflexionsgesetz berechnet, das Licht in verschiedene Richtungen gestreut wird. Da man in der Realität keine ideale Reflexion vorfindet, wird

die unvollkommene Reflexion auch real spiegelnde Reflexion genannt. Die betragsmäßig höchste Reflexion findet in die Richtung statt, die das Reflexionsgesetz vorgibt. Je größer der Winkel α zwischen der Betrachtungsrichtung und der Reflexionsrichtung ist, desto kleiner ist die gespiegelte Lichtintensität (vgl. Abbildung 3, die Richtung r gibt die Richtung an, die durch das Reflexionsgesetz entsteht, v gibt eine bestimmte interessierende Richtung an).

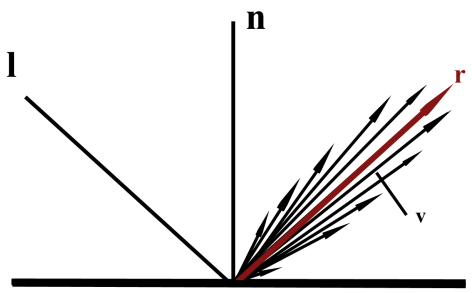


Abbildung 3: unvollkommene Reflexion [3, Seite 273]

Es wird also die Abhängigkeit des Abfalls der Intensität abhängig von α modelliert, welches der Winkel zwischen r und v ist. Der Phong-Beleuchtungsterm ergibt sich zu:

$$I_S = I_l r_S \max\left(0, \langle r, v \rangle\right)^n$$ **(Formel 3: Phong-Beleuchtungsterm)**

Dabei wird der Winkel zwischen r und v wieder durch das Skalarprodukt ausgedrückt. I_S steht für die Intensität des spiegelnden reflektierten Lichtes in Richtung v und I_l für die Intensität des von der Lichtquelle einfallenden Lichtes. Der spiegelnde Reflexionskoeffizient r_S stellt den Anteil des einfallenden Lichtes ein. Der Spiegelungsexponent n simuliert den Perfektionsgrad der Oberfläche zwischen 1-1000 (für große Werte von n entstehen scharfe kleine, für kleine Werte von n großflächige Highlights).

Da in den beiden genannten Modellen die Rückseiten nicht mit berücksichtigt werden, würden diese schwarz aussehen. Das hat aber keinen Bezug zur Realität und deshalb muss eine Hilfskonstruktion hinzugenommen werden: Die ambiente Beleuchtung.

Die ambiente Beleuchtung stellt eine allgemeine Hintergrundbeleuchtung dar, die man sowohl beim Lambertschen, als auch beim Phong-Modell hinzu addieren muss. Der ambiente Beleuchtungsterm lautet:

$$I_a = I_A r_a$$ **(Formel 4: ambienter Beleuchtungsterm)**

Wobei I_A die ambiente Beleuchtung und I_a die durch die ambiente Beleuchtung entstehende Intensität am Oberflächenpunkt darstellen. Der ambiente Reflexionskoeffizient r_a stellt wieder ein, wie stark die ambiente Beleuchtung wirkt.
Eine Version für monochrome Modelle für das Phong Beleuchtungsmodell lautet also:

$$\mathbf{I = I_a + I_d + I_S} = I_A r_a + I_l r_d \max\left(0, \langle l, n \rangle\right) + I_l r_S \max\left(0, \langle r, v \rangle\right)^n$$ **(Formel 5: Phong-Beleuchtungsmodell)**

In der Formel werden die Intensitäten des diffus reflektierenden, des spiegelnden Lichtes und die Intensität der ambienten Beleuchtung zusammen geführt.
Für farbige Modelle wird eine separate Betrachtung für I_R, I_G und I_B, also für die Intensität jeder Grundfarbe, durchgeführt. [3, Seite 278ff]

2.3 Schattierungsverfahren

Nachdem die Vertices beleuchtet wurden, müssen anschließend die Flächen der Polygone gefärbt werden. Dies wird mit Hilfe der Schattierungsverfahren ermöglicht. Schattierungsverfahren sind einfach und schnell und daher gut geeignet für Echtzeitrendering.

2.3.1 Flatshading

Das Flatshading ist das einfachste Schattierungsverfahren. Es wird an einem Punkt des Polygons die Farbe mit Hilfe einer Beleuchtungsberechnung an der Flächennormalen berechnet und diese Farbe auf das gesamte Polygon angewendet. Bei dem Verfahren treten Farbsprünge an den Übergängen der Facetten (der Begriff Facette wird für ein Polygon verwendet, welches ein Teil einer Oberfläche eines Objektes ist. [3, Seite 301]) auf, was die einzelnen Facetten hervorhebt und der Oberfläche ein kantiges Aussehen verleiht. (vgl. Abbildung 5 links).

2.3.2 Gouraud-Shading

Mit den störenden Sprüngen, die zwischen den Facetten beim Flat-Shading auftreten, beschäftigte sich Henri Gouraud. Wurde beim Flat-Shading nur eine Flächennormale berechnet, so befasst sich das Gouraud-Shading mit den Normalenvektoren, die an den Vertices zur Beleuchtungsberechnung benutzt werden. Diese Normalenvektoren werden durch eine Mittelung der Flächennormalen aller Flächen, die an einem Eckpunkt liegen, gemittelt und stehen so auf keine der Facetten senkrecht (eine Ausnahme ist der Fall, dass alle Facetten in einer Ebene liegen).

An den ermittelten Normalenvektoren wird dann die Beleuchtungsberechnung wie gehabt durchgeführt.

Um anschließend die Fragmentfarben in der Fläche ermitteln, werden entlang der Facettenkanten die Fragmentwerte durch lineare Interpolation[2] berechnet. In den Flächen werden anschließend die Fragmentfarben mit dem so genannten scan-line-Algorithmus[3] ermittelt und mit Hilfe der linearen Interpolation zwischen den vorher interpolierten Fragmenten auf den Facettenkanten eingefärbt.

Dies ist eine große Verbesserung im Bezug zum Flatshading (vgl. Abbildung 5 Mitte).

Doch fällt bei der Berechnung mit dem Gouraud-Shading ein Glanzlicht so auf das Objekt, dass es sich innerhalb einer Facette befindet, wird es nicht mit berechnet, da nur die Beleuchtung an den Eckpunkten der Facette berücksichtigt werden. Fällt es andererseits so auf das Objekt, dass ein Vertex beleuchtet wird, so wird gleich das ganze Polygon mit berechnet, und das Licht erscheint sternförmig. [5, Seite 187ff]

---

[2] Bei der linearen Interpolation wird aus gegebenen Punkten eine Kurve ermittelt, wobei die Punkte auf der Kurve liegen müssen. Im einfachsten Fall handelt es sich um zwei Punkte, die durch eine Gerade verbunden werden. [4]

[3] Beim scan-line-Verfahren wird eine horizontale Line von oben nach unten über das Polygon bewegt. Dafür werden als erstes die Kanten des Polygons nach ihren größten y-Werten und anschließend die scan-line vom größten y-Wert zum Kleinsten bewegt. Für jede Position an der scan-line wird eine Liste der Polygonkanten ermittelt, deren Schnittpunkte berechnet und nach x-Werten sortiert werden. Nun kann jeder Bildpunkt, der sich auf der scan-line und innerhalb des Polygons befindet angezeigt werden.[6]

2.3.3 Phong-Shading

Die Berechnung der Glanzlichter auf den Facetten der Objekte wird im Phong-Shading optimiert. In diesem Verfahren wird für jedes Pixel ein Normalenvektor bestimmt. (vgl. Abbildung 4)

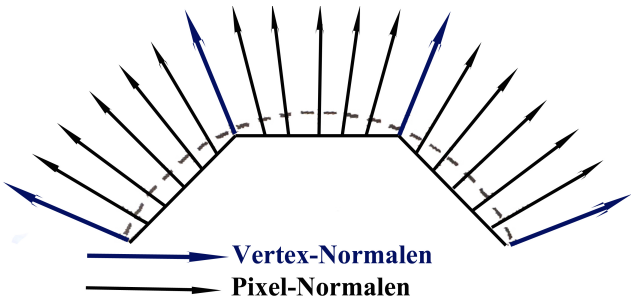


Abbildung 4: Pixel - Normalen – Interpolation [5, Seite 195]

Als Vertexnormalenvektoren nutzt man die Normalenvektoren, die auch schon im Gouraud-Shading ermittelt wurden. Die Pixelnormalenvektoren werden entlang der Facettenränder durch lineare Interpolation und innerhalb der Facette durch den scan-line-Algorithmus bestimmt.

Nachdem man für jedes Pixel einen Normalenvektor ermittelt hat, wird die Beleuchtungsberechnung an jedem Pixel durchgeführt. In der folgenden Grafik sieht man, dass nun die Helligkeitssprünge an den Polygonkanten des Gouraud-Shading beseitigt wurden. Das Phong-Shading ist somit das qualitativ hochwertigste aber auch rechenaufwendigste Verfahren. [5, Seite 195ff]

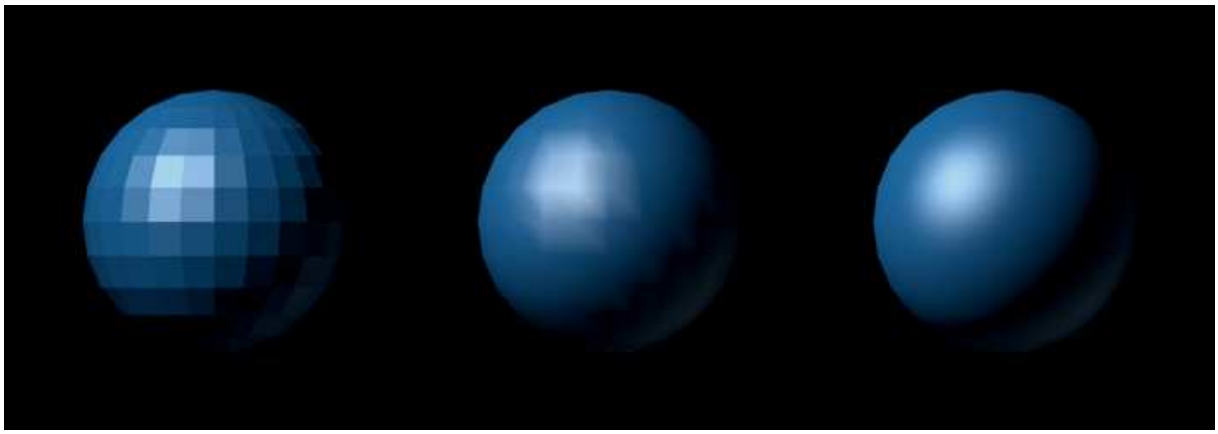


Abbildung 5: Shading-Modelle: Flat-, Gouraud- und Phong-Shading [7]

2.4 Texturmappingverfahren

Nachdem über die Beleuchtungs- und Schattierungsverfahren jedem Pixel eine Farbe zugeordnet werden kann, kann dies mit verschiedenen Texturmappingverfahren erweitert werden.

2.4.1 Abbildung einer Textur auf einem Objekt

Bei dem ursprünglichsten aller Texturmappingverfahren, geht es darum, dass ein zweidimensionales Bild auf die Oberfläche eines Objektes gelegt werden soll. Dies kann mit einer Tapete verglichen werden, die auf ein Objekt tapeziert wird. Die Tapete ist dabei die Textur.

Eine Textur wird in einem zweidimensionalen orthogonalen Koordinatensystem dargestellt, abhängig von zwei Veränderlichen, wobei oft u und v verwendet werden, die beide im Intervall [0,1] liegen. Um die Textur auf ein Objekt zu bringen, benötigt man einen Zusammenhang zwischen den Texturkoordinaten und den Objektkoordinaten des zu texturierenden Objektes. Die Textur muss dann in den dreidimensionalen Objektraum abgebildet werden.

Texturkoordinaten beliebiger Facetten werden ähnlich wie beim Gouraud-Shading über Interpolation ermittelt. Es werden also nur die Texturkoordinaten der Eckpunkte ermittelt und die dazwischen liegenden Fragmente über die Textur interpoliert. Wollte man ein Objekt ohne das Texturmapping mit einem Bild belegen, so müsste die Oberfläche dieses Objektes so fein strukturiert werden, dass für jedes Pixel der Textur mindestens einen Vertex zur Verfügung steht. Dies würde bedeuten, dass das Objekt sehr viele Polygone aufweisen muss und somit wieder ein großer Rechenaufwand entstehen würde. In folgender Abbildung wurden auf einem Rechteck drei Punkte bestimmt, welche die Eckpunkte der Textur auf dem Objekt festlegen. Zwischen den Punkten wurde interpoliert und die Textur wurde zusätzlich in u- und v-Richtung wiederholt. [3, Seite 297ff]

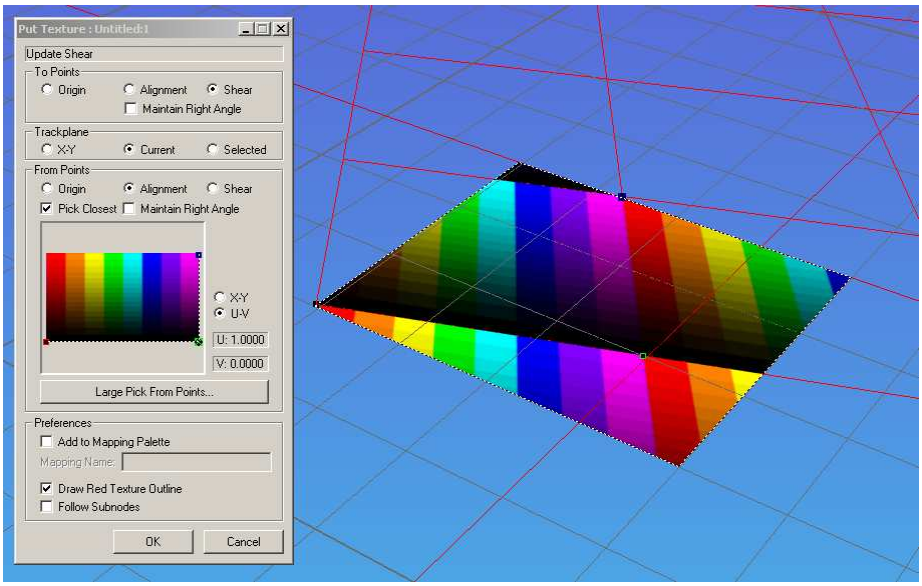


Abbildung 6: Texturierung eines Rechtecks mit dem Programm „MultiGen Creator“ erstellt

2.4.2 Abbildung einer Textur auf ein Objekt über eine Zwischenfläche

Um eine Textur auf einem aufwändigeren Objekt abzubilden, kann sie zuvor auf ein Objekt, welches als Zwischenablagefläche dient, abgebildet werden. Dafür werden einfach zu texturieren Objekte genutzt, wie z.B. ein Würfel, eine Kugel oder ein Zylinder. Anschließend wird auf das Objekt projiziert. Die Abbildung auf die Zwischenflächen wird S-Mapping (Surface-Mapping) und die Projektion auf das Objekt O-Mapping genannt. Das S-Mapping auf eine Kugel wird im Folgenden erläutert, da diese Rechnungen später in der Arbeit angewandt wird. Sobald die Textur auf einer Kugel liegt, kann für das O-Mapping z.B. der Strahl senkrecht zur Zwischenablage auf das Objekt projiziert werden, so dass auf dem Schnittpunkt des Strahls mit dem Objekt die Farbe des Pixels in Objektkoordinaten vorliegt. [8]

2.4.2.1 Abbildung einer Textur auf der Zwischenfläche Kugel

Bei diesem Mapping-Verfahren wird die Textur auf eine Kugel projiziert. Die Kugel liegt in dreidimensionalen kartesischen Koordinaten vor, worin ein Punkt definiert wird als P(x,y,z). Die Textur liegt in zweidimensionalen u,v-Koordinaten vor. Für die Umrechnung wird der Weg über Kugelkoordinaten benötigt.

Ein Punkt in Kugelkoordinaten wird angegeben durch P(r, φ, λ). R ist dabei der Radius der Kugel, φ beschreibt den Längengrad (dies ist der Winkel zwischen der z-Achse und der vertikalen Position des Vektors P) und ist definiert in dem Intervall [0, 2π]. λ beschreibt den Breitengrad (der Winkel zwischen der z-Achse und dem Vektor P) und ist im Intervall [0, π] definiert. Die Kugelkoordinaten werden in Abbildung 7 verdeutlicht.

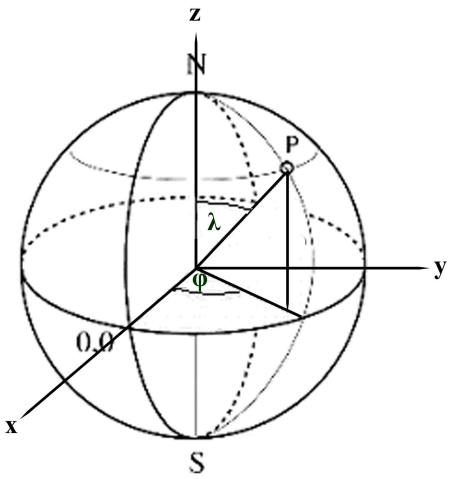


Abbildung 7: Kugelkoordinaten [9]

Die Umrechnung von den kartesischen Koordinaten in Kugelkoordinaten ergibt sich durch folgende Formeln:

$$r = \sqrt{x^2 + y^2 + z^2}$$ **(Formel 6: Radius)**

$$\varphi = a\tan 2(y,x) = \begin{cases} \arccos\dfrac{x}{\sqrt{x^2+y^2}} & für \quad y \geq 0 \\ \arccos\dfrac{x}{\sqrt{x^2+y^2}} - 2\pi & für \quad y < 0 \end{cases}$$ **(Formel 7: Längengrad)**

$$\lambda = \arccos\frac{z}{r}$$ **(Formel 8: Breitengrad)**

Der Punkt P, der nun umgewandelt wurde von P(x,y,z) in P(r, φ, λ), muss anschließend noch in Texturkoordinaten, also P(u,v) umgewandelt werden. Der Radius wird für diese

Zwecke normalisiert und die Intervalle [0, 2π] bzw. [0, π] sollen auf das Intervall [0,1] abgebildet werden. Folgende Formeln ergeben sich anschließend für die Texturkoordinaten:

$$u = \frac{\lambda}{\pi} = \frac{\arccos \frac{z}{r}}{\pi}$$

da r normalisiert wurde:

$$u = \frac{\lambda}{\pi} = \frac{\arccos z}{\pi}$$ **(Formel 9: Texturkoordinate1)**

$$v = \frac{a \tan 2(y, x)}{2\pi}$$ **(Formel 10: Texturkoordinate2)**

Nun wurde eine Abhängigkeit der Texturkoordinaten zu den kartesischen Koordinaten der Kugel geschaffen. So kann jede beliebige Textur im Intervall [0,1] auf eine Kugel projiziert werden. [10]

2.4.3 Environment Mapping

Bei spiegelnden Objekten wäre es schön, die Umgebung in den Objekten gespiegelt zu sehen. Für aufwändige und qualitativ hochwertige Berechnungen kann das Ray-Traycing-Verfahren genutzt werden. Doch da dies in Echtzeit nicht einsetzbar ist, kann man sich einer Hilfskonstruktion bedienen: dem Environment Mapping.

Dafür muss als erstes ein Foto der Umgebung gemacht, und als Environment-Map gespeichert werden. Der Strahl, der von der Kamera auf das Objekt fällt muss anhand des Normalenvektors dieses Punktes reflektiert werden. Der Reflexionsstrahl trifft einen Punkt in der Environment-Map, der auf dem Objekt abgebildet wird. Der reflektierte Strahl wird anhand des Reflexionsgesetzes[4] berechnet. [1]

Zwei bekannte Verfahren sind:

Das Sphere-Environment-Mapping und das Cubic-Environment-Mapping. Das Sphere-Environment-Mapping verwendet eine einzelne Textur als Spiegeltextur, die Sphere-Map bezeichnet wird. Die Grundidee besteht darin, eine ideal verspiegelte Kugel aus weiter Ferne zu fotografieren. In dieser Kugel spiegelt sich die gesamte Umgebung zu den Rändern hin verzerrt. Das Hauptproblem dieses Verfahrens ist, dass die Reflexion nur für

[4] Beim Reflexionsgesetz gilt, dass der Einfallswinkel (zwischen dem einfallenden Strahl und dem Normalenvektor des Punktes) gleich dem Ausfallswinkel (zwischen dem reflektierten Strahl und dem Normalenvektor des Punktes) ist.

einen Blickwinkel gilt. Bewegt sich der Beobachter um das Objekt herum, so sieht er nicht die andere Seite der Umgebung, sondern dieselbe, wie vorher.

Abbildung 8: Sphere-environment-Mapping [11]

Dieses Problem ist beim Cubic-Environment-Mapping beseitigt. Hier wird nicht nur eine Textur verwendet, sondern sechs; für jede Seite eines Würfels eine. Dieser Würfel dient als Umgebung um das Objekt. Die Textur wird so erstellt, dass alle sechs Seiten eines Raumes fotografiert werden. Abbildung 9 zeigt ein Beispiel.

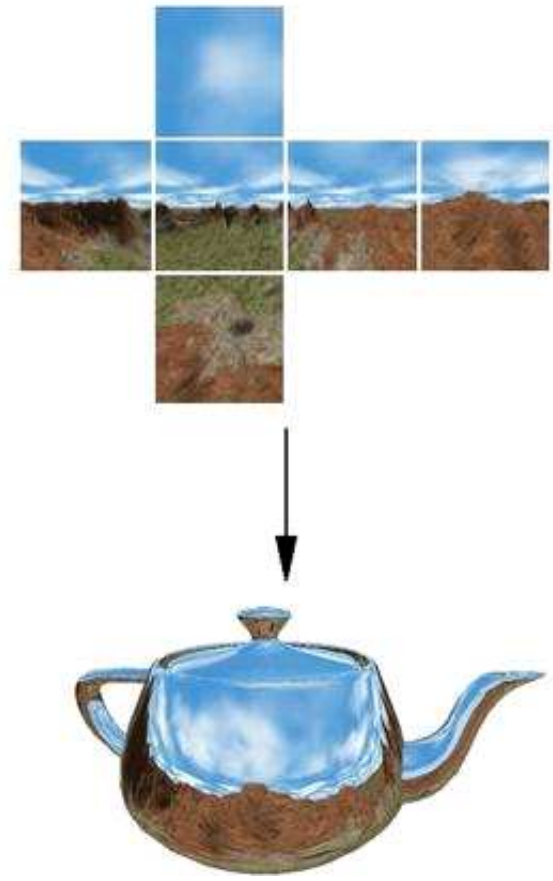


Abbildung 9: Cube-Environment-Mapping [12]

Der Reflexionsvektor zeigt bei diesem Verfahren auf eine der sechs Texturen und sobald man sich um das Objekt herum bewegt, wird auch die Rückseite der Umgebung betrachtet. Bei der Erstellung der Texturen ist zu beachten, dass sie an den Übergängen genau zusammen passen müssen. [5, Seite 258ff]

2.4.4 Multitexturierung

Bisher wurde immer nur der Prozess beschrieben, bei dem eine Textur auf ein Objekt angewandt wird. Dies war entweder eine auf dem Objekt abgebildete, oder eine Umgebungstextur. Sobald für mehr Vielfalt zwei Texturen auf ein Objekt anwendet werden sollen, ist dies mathematisch gesehen eine Kombination der beiden Textur-Fragmentfarben an einem Punkt. Es können mathematische Operationen, wie eine Multiplikation, Addition oder Interpolation angewandt werden. [5, Seite 252]

2.4.5 Texturfilter

Die Textur als Funktion abhängig von u und v darzustellen, bedeutet, dass die Textur als ein Array mit diskreten Werten dargestellt wird. Daraus folgt, dass die u- und v-Werte durch Rundung ermittelt werden, wenn ein Wert zwischen zwei diskreten Werten liegt und wird „nearest-neighbor-Methode“ genannt. Dies ist eine schnelle, aber sehr ungenaue Filter-Methode. Wenn sich zwei benachbarte Werte sehr stark unterscheiden, bewirkt eine leichte Kamerabewegung, dass in der Darstellung zwischen den Werten hin und her gewechselt wird, so dass ein Flackern, oder Aufblitzen entsteht,

Wenn ein Pixel mehrere Texel (Pixel der Textur) überdeckt, muss die Textur verkleinert werden, dieser Vorgang wird Minification genannt. Bedeckt ein Pixel nur den Teilbereich eines Texels, ist eine Magnification notwendig, wobei die Textur vergrößert wird. Dies kann über unterschiedliche Filteroperationen realisiert werden, eine der einfachsten ist die lineare Interpolation.

Entfernt sich die Kamera von einem texturierten Objekt, so wird die auf dem Objekt angebrachte Textur mit dem Objekt verkleinert. Dafür hilft die Mip-Maptechnik. Hier werden von jeder Textur vorgefilterte Texturen in unterschiedlichen Größen abgespeichert. Es wird dann immer zwischen den jeweiligen Größen umgeschaltet. [3, Seite 306f]

2.5 Rendering-Pipelines

Rendering-Pipelines stellen dar, wie aus gegebenen dreidimensionalen Objekten, einer virtuellen Kamera, Lichtquellen, Shadingsystemen und Texturen ein zweidimensionales Bild gerendert wird.

2.5.1 Fixed-Function-Pipeline

Diese Pipeline ist eine Folge von festen Abläufen, die teilweise hintereinander und teilweise parallel ablaufen. Abbildung 10 zeigt einen groben Aufbau der Pipeline in ihre drei Stufen Applikation, Geometrie und Rasterung. Wie zu sehen ist, sind die Stufen Geometrie und Rasterung selbst eigene Pipelines.

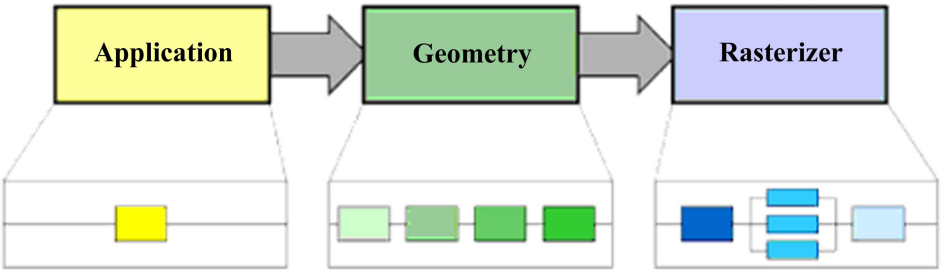


Abbildung 10: Pipeline Übersicht [13, Seite 13]

Die ***Applikations-Stufe*** beinhaltet auf der CPU ausgeführte Programme, um den Renderprozess auf der GPU zu steuern. Prozesse, wie Kollisionserkennung, Aufnahme von Inputs aus Quellen, wie der Tastatur- oder Maus oder Texturanimationen werden hier ebenso, wie Algorithmen zur Prozessbeschleunigung durchgeführt.
Die Ergebnisse dieser Stufe sind die zu rendernden Primitive.

Die zweite Stufe, die ***Geometrie-Stufe***, ist aufgeteilt in 5 Unterstufen (vgl. Abbildung 11). Diese Stufe ist für die meisten Vertex- und Polygon-Operationen verantwortlich.

Abbildung 11: Geometrie-Stufe [13, Seite 16]

Die *Model- und View-Transformation* beschäftigt sich damit, das Objekt, welches im Objektkoordinatensystem entsteht, in das Kamerakoordinatensystem umzuwandeln. (Es handelt sich hierbei um Transformationen der Vertices und der Normalenvektoren.) Dafür muss das Objekt als erstes mit Hilfe einer Transformationsmatrix von seinem Objektkoordinatensystem in das Weltkoordinatensystem und von dort aus mit einer weiteren Matrix in das Kamerakoordinatensystem (auch Viewspace oder Eyespace genannt) transformiert werden.

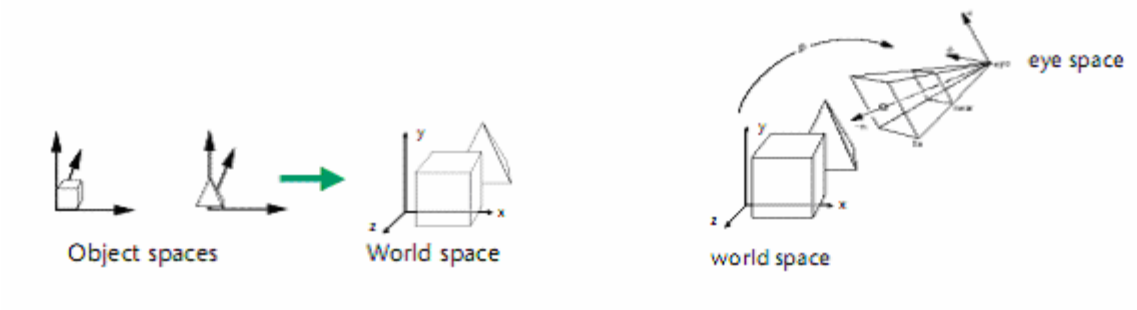


Abbildung 12: Koordinatentransformationen [1]

Transformationen werden im Zusammenhang mit der genutzten Software in Kapitel 3.4 noch ausführlicher erklärt.

Beim *Vertex-Shading* werden die Vertices dem Shading unterzogen. Shading bedeutet Lichteffekte und Materialien eines Objektes miteinander zu verrechnen. Dafür stehen für einen Vertex mehrere Parameter zur Verfügung, wie z.B. deren Position, Farbe oder Normalenvektoren. Als Ergebnis des Vertex-Shadings können Shadingdaten, wie z.B. Farben oder Texturkoordinaten vorliegen. Diese werden der Raster-Stufe übergeben, damit sie interpoliert werden können.

In der *Projektions-Stufe* wird das Sichtvolumen der Kamera, welches sich bisher noch über eine Pyramide definiert hat, in ein kanonisches Sichtvolumen umgewandelt. Das kanonische Sichtvolumen definiert sich über einen Einheitswürfel.

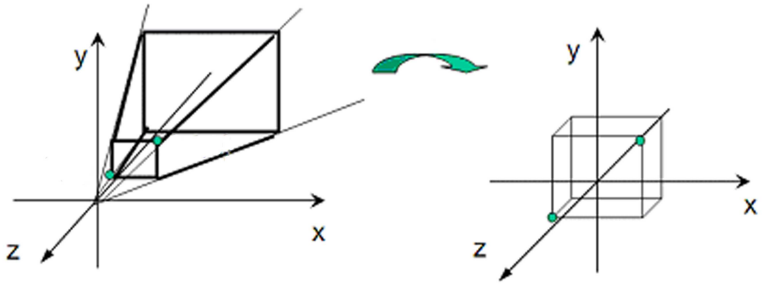


Abbildung 13: Kanonisches Sichtvolumen [14]

Da nur die Primitive, die im Sichtvolumen auch zu sehen sind, der Rasterisierungsstufe weiter gegeben werden müssen, ist das Clipping notwendig.

Beim *Clipping* werden die Primitive, die sich nur teilweise im Sichtvolumen befinden so abgeschnitten, dass neue Vertices entstehen, die nun nicht mehr außerhalb des Sichtvolumens liegen. Abbildung 14 demonstriert diesen Vorgang an dem gelben Dreieck. Der äußere rechte Vertex liegt nicht mehr im Sichtvolumen und wird so abgeschnitten, dass zwei neue Vertices entstehen. Das rote Dreieck in der Grafik wird aussortiert und so auch nicht mit gerendert. Dieser Prozess heißt culling.

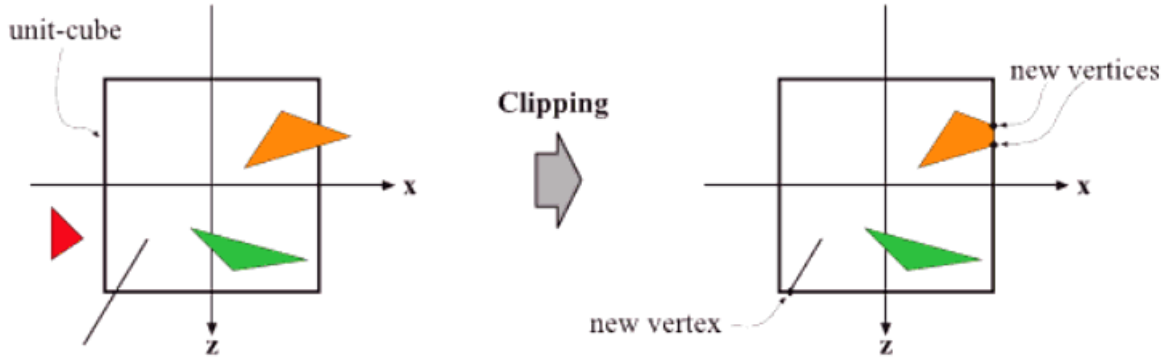


Abbildung 14: Clipping [13, Seite 20]

Als letzte Stufe der Geometrie Stage folgt das *screen mapping*. Dieser Stufe werden nur die geclippten Primitive und diese, die komplett innerhalb des Sichtvolumens liegen, übergeben. Die Koordinaten der Primitiven sind bis hier immer noch dreidimensional (besitzen also eine x-, y- und z-Koordinate). In dieser Stufe werden die x- und y-Koordinaten in einem Fenster abgebildet von (x_1, y_1) bis (x_2, y_2) und die z-Koordinate in einem z-Puffer gespeichert. Die z-Koordinate stellt die Tiefe eines jeden Punktes im Raum dar und wird später noch benötigt.

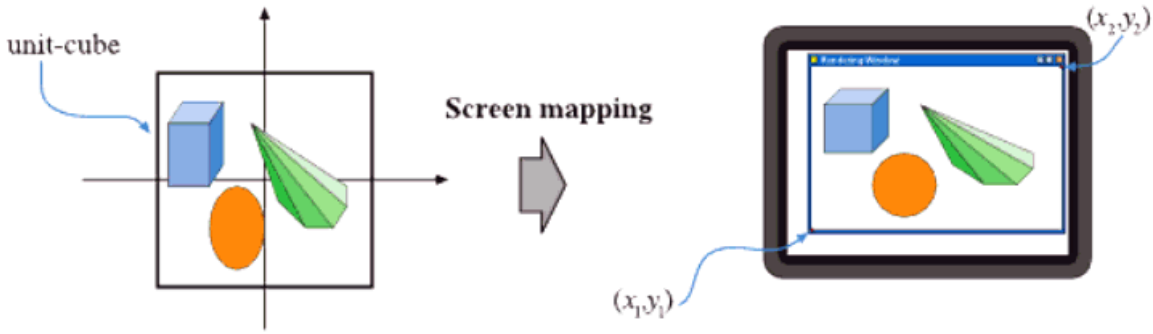


Abbildung 15: screen mapping [13, Seite 20]

Die letzte Stufe der Pipeline ist die ***Rasterungs-Stufe***. Am Eingang liegen die zweidimensionalen Koordinaten vor. Als Ausgang aus dieser Stufe sollen die Farben der einzelnen Pixel des Ausgabegerätes vorliegen. Die Rasterungs-Stufe stellt wieder eine eigene Pipeline dar. Diese Teilabschnitte lassen sich parallelisieren (vgl. Abbildung 10).

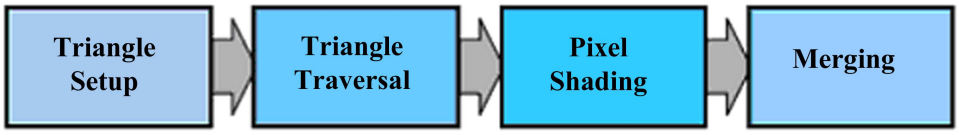


Abbildung 16: Rasterungs-Stufe [13, Seite 22]

Beim *Triangle Setup* werden die Daten für die Oberfläche der Dreiecke berechnet. Diese Daten werden für den nächsten Schritt und für die Interpolation der Shadingdaten aus der Geometrie-Stufe benötigt.

Beim *Triangle Traversal*, auch scan conversion genannt, werden Fragmente, die innerhalb eines Dreiecks liegen gesucht, damit sie später gefüllt werden können.

Die *Pixelshading-Stufe* berechnet mit Hilfe der Shadingdaten aus der Vertexshading-Stufe für jedes Fragment, das in der Stufe Triangle Traversal innerhalb eines Dreiecks definiert wurde, eine Farbe.

Beim *Merging* werden alle Fragmente getestet, ob sie von der Kamera aus gesehen sichtbar sind. Mit Hilfe des z-Puffers, der schon in der screen mapping-Stufe erwähnt wurde, wird der Depth-Test durchgeführt. Der z-Wert eines Fragmentes gibt dessen Tiefe im Bild an. In diesem Test wird geprüft, ob das aktuelle Fragment vor oder hinter einem anderen liegt, welches die gleichen x- und y-Koordinaten hat. Liegt es davor, so wird der Farbwert übernommen, liegt es dahinter, verfällt es.

Der Farbwert eines Fragmentes besteht aus vier Komponenten: RGB und A, die im Color-Puffer gespeichert werden. Neben den drei Grundfarben RGB wird der Wert A für die Transparenz des Fragments gespeichert, wobei Null transparent und eins opak bedeutet. Bei dem Alphatest wird geprüft ob, und mit welcher Intensität ein Pixel gezeichnet wird – abhängig von der Transparenz des Fragments.
Im Stenciltest (zu deutsch Schablone) wird festgestellt, ob ein Fragment im Bereich einer möglicherweise vorher erstellten Schablone (über den sichtbaren Bereich) liegt.
Werden alle Tests bestanden, so wird die Farbe des Fragments im Color-Puffer gespeichert. Diese Tests nennen sich Rasteroperationen und sind in OpenGL und Direct3D (vgl. Kapitel 2.1.3) implementiert. [13, Seite 11ff]

2.5.2 Programmable-Function-Pipeline und die Einführung von Shadern

Auf die Durchführung der Stufen der Fixed-Function-Pipeline hatte ein Grafikprogrammierer lange keinen Einfluss. Die Algorithmen waren fixiert. Besondere oder individuelle Wünsche und Effekte, die nicht zur Pipeline-Implementierung passten, blieben der langsamen Software-Lösung auf der CPU vorbehalten.
Im Jahr 2001 wurde Programmierern der Zugang und so die Einflussnahme auf die in Hardware gegossenen Pipeline-Stufen durch das Laden kleiner Assembler-Programme in die Grafikkarte ermöglicht. Diese kleinen in Maschinensprache geschriebenen Programme werden Shader genannt und bezeichnen ein Stück Software, welches an die Stelle eines Teils der Hardware-Pipeline tritt. Wird bei einem Objekt kein Shader definiert, so durchläuft es die Fixed-Function-Pipeline. [3, Seite 294ff]

2.5.3 Arten von Shadern

In Abbildung 17 ist die Implementierung der Pipeline in der GPU zu erkennen. Die grünen Stufen sind komplett programmierbar, die gelben sind konfigurierbar, aber nicht programmierbar und die blauen Stufen sind in ihrer Funktion fixiert.

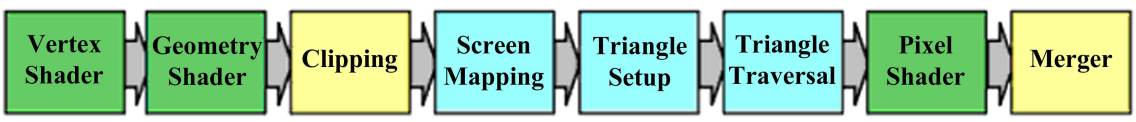


Abbildung 17: programmable pipeline [13, Seite 30]

In der Abbildung ist ebenfalls zu erkennen, dass es drei verschiedene Arten von Shadern gibt, die Vertex-, Geometry- und Pixelshader. [13, Seite 30]

Der Vertex-Shader bekommt eine Reihe von Parametern als Input, wie Eckpunkte und Normalenvektoren in Objektkoordinaten, Materialparameter und Texturparameter der Vertices. Dies können z.B. OpenGL Zustände oder Vertex-Parameter sein. So kann der Shader folgende Funktionen der fixed-function-pipeline ersetzen:

- Vertex- und Normalentransformation ins Kamerakoordinatensystem
- Normalisierung
- Per-Vertex-Beleuchtungsberechnung
- Generierung und/oder Transformation von Texturkoordinaten

Der Fragment-Shader bekommt als Input die Parameter, die der Vertex-Shader als Output heraus gibt. Diese Parameter sind nun interpoliert. Gibt ein Vertex-Shader die Farbe eines Vertex aus, so bekommt der Fragment-Shader als Input für ein Fragment die Farbe, welche über die Vertex-Daten interpoliert wurde. Als Output gibt der Fragment-Shader die Farbe für den Pixel aus. Dazu ersetzt er die Operationen auf die interpolierten Werte in der fixed-function-pipeline. Ein Fragment-Shader hat keinen Zugriff auf benachbarte Fragmente, sondern bezieht sich immer nur auf das eine aktuelle Fragment. [1]

Mit DirectX 10 wurde zu den beiden Shadern noch der Geometry-Shader hinzugefügt. Der Input eines Geometry-Shaders ist ein einzelnes Objekt und dessen Vertices. Dieses Objekt besteht üblicherweise aus Primitiven, die vom Geometry-Shader zerstört oder neu erschaffen werden können. Dieser Shader ist optional und wird in dieser Arbeit nicht genutzt. [13, Seite 40ff]

2.6 High-Level-Shader-Sprachen

Wie in Kapitel 2.5.2 erwähnt, wurden anfangs die Shader als Assembler-Programme in die Grafikkarte geladen. Diese Technik wurde zwar von OpenGL und Direct3D unterstützt, aber von wenigen Programmierern umgesetzt, da sie in einem Assembler Code geschrieben werden mussten. Die unübersichtliche Sprache war verantwortlich dafür, dass der Kreis der Grafikprogrammierer, die dazu bereit waren Shader zu programmieren, sehr

klein war. Zur Erleichterung wurden seit 2002 verschiedene Hochsprachen eingeführt, die diese Assemblersprachen ersetzten und eine strukturiertere Programmierung ermöglichten.
Die Arbeit mit den Shadersprachen ist anders, als mit selbst erstellten Programmen. Denn ein Shader ist nur ein Teilstück der programmierbaren Pipeline. Das Shaderprogramm entspricht somit einem Knoten, durch den Daten zur Bildberechnung fließen. So werden dem Shader ständig Daten übergeben und dieser muss ständig Ausgaben an den nachfolgenden Knoten liefern.
Es gibt verschiedene Shadersprachen, wie z.B: Cg (C for graphics), HLSL (High-Level Shader Language) oder GLSL (OpenGL Shading Language). Im Folgenden wird die Sprache Cg und deren Erweiterung CgFX genauer erläutert, da diese in der Arbeit benötigt werden. [3, Seite 294]

2.6.1 Cg = „C for graphics“

Die High-Level-Shader-Sprache Cg (vgl. [16]) wurde von NVIDIA in enger Kooperation mit Microsoft entwickelt und 2002 vorgestellt. Sie kann unabhängig von der Grafikprogrammierschnittstelle, also OpenGL und Direct3D, eingebunden werden und ist damit ebenfalls unabhängig vom Betriebssystem. Es ähnelt von der Syntax her, der Programmiersprache C, ist aber speziell auf die Arbeit mit Grafiken angepasst.
Die Parameterübergabe an ein Cg-Programm erfolgt über zwei verschiedene Typen, die uniform- und die varying-Werte. Varying-Werte sind Werte, die sich von Pixel zu Pixel, oder von Vertex zu Vertex unterschieden, wie z.B. die Position, die Normale oder der Texturkoordinatensatz. Eingaben des Typs uniform dienen zur Übergabe von Parametern an das Programm. Sie ändern sich entweder sehr langsam oder gar nicht. Diese Parameter werden mit dem vorangestellten Codewort uniform gekennzeichnet.
Die Ausgabewerte eines Programms können auf zwei Wegen ausgegeben werden. Entweder man kennzeichnet sie mit dem vorangegangenen Codewort out, oder definiert einen Output-Container, mit dem alle Parameter ausgegeben werden, die an diesen gehängt werden.
Die Eingabeparameter werden von der Pipeline in das Programm übergeben, die Ausgabeparameter vom Programm an die Pipeline. So wird das Programm an die Pipeline gebunden. Dafür gibt es reservierte Cg-Codewörter, die mit einem Doppelpunkt und mit großen Buchstaben hinter die Werte geschrieben werden.

```
struct myinputs{
      float3 myPosition       :POSITION;
      float3 myNormal         :NORMAL;
      float3 myTangent        :TANGENT;
      float refractive_index  :TEXCOORD3;
};
```

Abbildung 18: Semantics [15, Seite 6]

Als Beispiel für diese so genannten Binding Semantics gibt es für den Vertex-Shader die Eingabewerte POSITION (Ortskoordinaten des Vertex), NORMAL (Normalenvektor des Vertex) oder TEXCOORD0 (Texturkoordinatensatz 0).
Bei Fragment-Shadern ist nur das Codewort COLOR als Output zulässig, da, wie schon erwähnt, der Ausgabewert aus einem Fragment-Shader nur eine Farbe sein kann.
Der Name „C for grafics“ zeigt, dass diese Sprache zwar an C angelegt ist, diese aber für die spezielle Anwendung in der Computergrafik erweitert wurde. Es gibt folgende Datentypen, die die Arbeit mit Computergrafiken erleichtern. So findet man in Cg 2-, 3-, und 4- dimensionale Vektortypen, die mit *float2*, *float3* und *float4* gekennzeichnet sind. Auch Matrizen können initialisiert werden. *Float4x4* ist z.B. eine Matrix mit vier Spalten und vier Zeilen. Eine weitere Besonderheit sind die Texturdatentypen. Es gibt 1-, 2-, 3-dimensionale und Cube Map-Texturen. Die Initialisierung erfolgt mit *sampler1D*, *sampler2D*, *sampler3D* und *samplerCUBE*.
Zusätzlich zu den Matrizen und Vektortypen gibt es spezielle mathematische Operationen, die die Arbeit mit Vektoren und Matrizen erleichtern. Zu den Vektoroperationen gehören u.a. die Normierung von Vektoren über „normalize()“, das Skalarprodukt „dot()“, das Vektorprodukt „cross()“ oder die Berechnung von Reflexions- und Brechungsvektoren über die Methoden „reflect()“ und „refract()“. Die Aufrufe „mul()“, „transpose()“ oder „determinant()“ berechnen die Multiplikation zweier Matrizen, oder die Transponierte bzw. Determinante. Oft sind die Funktionen überladen. So kann z.B. das Skalarprodukt für 2-, 3-, und 4- dimensionale Vektoren berechnet werden.
Der Swizzle-Operator erlaubt es, auf einzelne Komponenten von Vektoren zu zugreifen. Implementiert man z.B. einen Vektor mit *float4 v = (4, 3, 2, 1);* und ruft mit Hilfe dieses Operators float *f = v.y;* auf, so erhält f den Wert 3. Die Werte in den Vektoren werden bezeichnet mit .xyzw oder .rgba. float f = v.g; würde also ebenfalls den Wert 3 zurückgeben. So kann man beliebig auf Vektordaten zugreifen. Andere Beispiele sind:
float2 f = v.xz;float3 f = v.xxx; *float4 f = rbag;*

Auch Methoden für Texturen, wie „tex2D()“, welche aus einer gegebenen Textur und einer Koordinate die Farbe eines Pixels zurückgibt, sind in Cg implementiert.
Da nicht alle Grafikkarten den Umfang eines Cg-Programms verarbeiten können – manche lassen z.B. nur Vertex- oder Fragment-Shader zu oder können nur einen Teil der mathematischen Operationen durchführen – deckt Cg diese Vielfalt der Grafikkarten durch so genannte Profile ab. Für verschiedene Hardware-Betriebssystem-API-Kombinationen legt das Cg Programm damit den zur Verfügung stehenden Sprachumfang fest. [5, Seite 198] und [16]

2.6.2 CgFX = Cg für Effekte

CgFX ist ein shading-effect-system, welches auf Cg aufgebaut wurde und einen kompletten Shading-Effekt beschreibt. Alles was für einen Shading-Effekt benötigt wird, ist so mit CgFX in einem File vorhanden. Zum Aufbau eines CgFX-Files gehören Annotations, Vertex-Shader, Fragment-Shader und Techniken. Fragment- und Vertex-Shader sind auch hier in Cg geschrieben.
Die Techniken in einem CgFX-File dienen dazu das Programm so zu kompilieren, dass es zu der entsprechenden Hardware und genutzten API passt. In Cg wird das Programm an die vorhandenen Bedingungen durch die Profile angepasst. Dies geschieht auch in CgFX, wobei hier die Profile in die Techniken geschrieben werden. Das ermöglicht ein Programm mit mehreren unterschiedlichen Techniken zu schreiben. Eine Technik kann dann auf der neuesten GPU aufbauen, wobei die andere Technik Möglichkeiten für eine ältere GPU enthält. In einer Technik wird das Profil für den jeweiligen Vertex- und Fragment-Shader aufgezeigt.
Annotations erleichtern dem Programmierer einen direkten Zugriff auf das Programm. In Maya entsteht dadurch direkt ein User Interface (UI). Bei dem Beispiel

```
float gKd <
      string UIWidget = „slider“;
      float UIMin = 0.0;
      float UIMax = 1.0;
      float UIStep = 0.01;
      string UIName = „Diffuse Strength“;
>= 0.9;
```

Abbildung 19: Annotations [18, Seite 8]

gibt es fünf Annotations und einen Default-Wert. Alle Annotations sind optional, sie müssen nicht gesetzt werden. Hier wird ein Schieberegler erstellt im Bereich von Null bis eins, welcher in den Abstufungen 0,01 gesetzt werden kann. *UIName* gibt den Namen auf dem User Interface an, und *UIWidget* die Art des Schiebereglers, wobei es neben dem „slider“ noch die Möglichkeiten „Color“ oder „None“ zu setzen gibt. Bei „Color“ muss mindestens ein dreidimensionaler Vektor eingegeben werden. Das User Interface sieht anschließend aus wie in Abbildung 20.

Abbildung 20: Color Bedienfeld

Bei boolean-Variablen wird automatisch ein Kästchen erstellt, indem ein Haken wahlweise gesetzt werden kann (vgl. Abbildung 21).

```
bool example = false;
```

Abbildung 21: erstellt ein Kästchen auf dem User Interface

Möchte man eine Textur in den Shader laden, so muss hier mehr beachtet werden als bei einem Vektor- oder float-Parameter.
Bei Texturen werden zwei Parameter benötigt, einmal die eigentlichen Daten der Textur und als zweites der so genannte Textur-Sampler. Der Sampler gibt die Texturdaten an den Shader weiter. Texturen können Annotations besitzen, Textur-Sampler besitzen Zustandsbeschreibungen, die aussehen wie Annotations, aber keine sind. Diese werden nicht so wie die Annotations in eckige Klammern, sondern in geschweifte Klammern gesetzt, wie man am Beispiel in Abbildung 22 erkennen kann.

```
Texture gEnvTexture <
      //These are annotations
      string ResourceName = „default_reflection.dds“;
      string UIName = “Environment”;
      string ResourceType = “Cube”;
>;

samplerCUBE gEnvSampler = sampler_state{
      //These are not Annotations
      Texture = <gEnvTexture>;
      MagFilter = Linear;
```

```
      MinFilter = Linear;
      MipFilter = Linear;
      AddressU = Clamp;
      AddressV = Clamp;
      AddressW = Clamp;
};
```

Abbildung 22: Textur Annotation und Sampler State [18, Seite 10]

RessourceName gibt den Namen einer möglichen Textur an, die als default-Textur geladen werden kann. *RessourceType* gibt an, ob es sich um einen *1Dsampler*, *2Dsampler*, *3Dsampler* oder *samplerCUBE* handelt. Als Texturfilter ist in allen drei Filtern die lineare Interpolation als Option gewählt. Andere Möglichkeiten in CgFX sind None, Point, Linear, Anisotropic, PyramidalQuad, und GaussianQuad. AdressU, V und W geben an, wie sich die Textur verhalten soll, wenn die zu texturierende Fläche größer als die Textur ist. Die gängigsten Möglichkeiten sind repeat, mirror, clamp und border (vgl. Abbildung 23). In CgFX sind zusätzlich Wrap, ClampToEdge, ClampToBorder, MirroredRepeat, MirrorClamp, MirrorClampToEdge, MirrorClampToBorder und MirrorOnce implementiert.

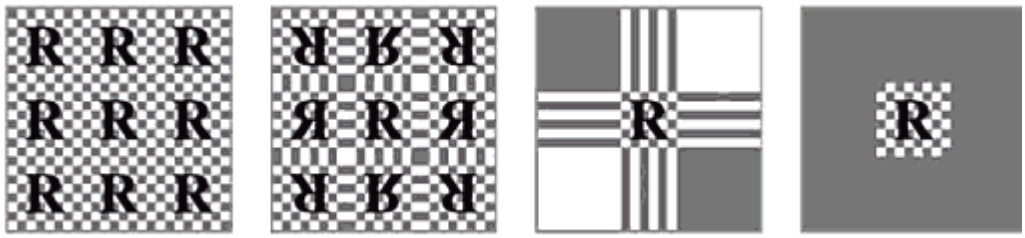


Abbildung 23: textur repeat, mirror, clamp und border [13, Seite 155]

Eine weitere Besonderheit von CgFX-Files ist die erweiterte Nutzung von Semantics. Hier können den Shadern zusätzliche Werte, die von der Softwareumgebung kommen, wie z.B. die Weltkoordinaten von Lichtern, Lichtfarben oder Transformationsmatrizen, übergeben werden. Diese Parameter werden an den Anfang des CgFX-Files geschrieben und sehen z.B. so aus:

float4x4 WorldXf : WORLD;

Diese Matrix beinhaltet die Transformation von Objekt in Weltkoordinaten. [17] und [18]

3 Genutzte Grafiksoftware

Zur Erstellung eines Szenenbildes wird im WDR Köln die Software Maya genutzt. Im virtuellen Studio wird mit der Software 3Designer gearbeitet.

3.1 Maya

Maya ist eine 3D-Modellierungssoftware, die von „Alias Wavefront“ entwickelt wurde und mittlerweile eine so feste Größe in der 3D-Grafikbranche geworden ist, dass es von Autodesk, der Hauptkonkurrenz, übernommen wurde.

So wurden z.B. Filme wie Avatar oder Computerspiele wie Ghostbusters mit Maya realisiert. Aber auch in der medizinischen Visualisierung, Architekturdesign, Sendegrafik, Industriedesign, bei visuellen Effekten oder Webdesign wird Maya verwendet.

Maya arbeitet mit einer Knoten-Architektur. Das bedeutet, dass alle Informationen in Maya miteinander verknüpft sind. Wie die einzelnen Knoten, so genannte Nodes, miteinander verknüpft sind, kann man im Hypergraph einsehen. Für ein einzelnes Objekt gibt es Knoten, wie z.B. die Position, die Form oder die Farbe.

Die Render-Nodes sind Texturen, Materialien, Lichter, Render-Werkzeuge oder Effekte, wie die CgFX-Shader. Da zwischen diesen Knoten viele Verknüpfungen bestehen, gibt es neben dem Hypergraph den Hypershade. Hier werden die Render-Nodes als Ikon dargestellt, man bekommt einen einfachen und schnellen Zugriff und kann sie editieren, verbinden oder neue erstellen. Die Render-Nodes können einzelnen Objekten zugewiesen werden, indem man das Objekt markiert und mit dem Befehl „assign to material“ dem Objekt zuweist.

Die einzelnen Netzwerke eines Knotens werden in einer Karteikarte angezeigt, wenn man einen Knoten auswählt. Hier sind auch die In- und Output-Connections mit Pfeilen aufgezeigt, die Maya automatisch in Abhängigkeit des Knotentyps erstellt. Connections können gelöscht, verschoben, anders verknüpft oder neu hinzugefügt werden.

Im Attribute Editor können die Eigenschaften eines Knotens verändert werden.

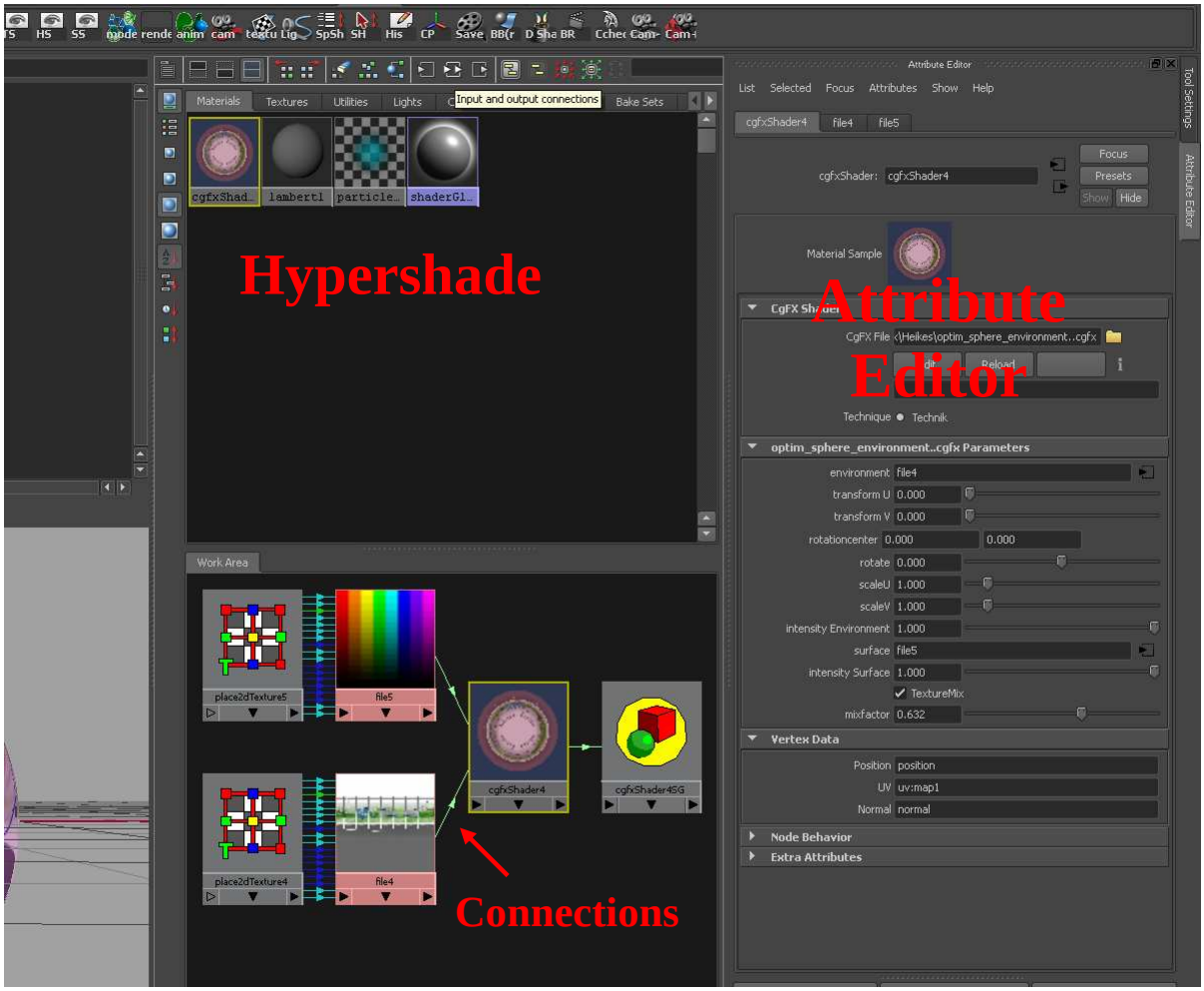


Abbildung 24: Maya

Maya hat vier integrierte Renderer: Maya Software, Maya Hardware, Vector Renderer und Mental Ray. Der Maya Software Renderer und Mental Ray ermöglichen außerordentliche Effekte in einer guten Auflösung und sind in ihrer Geschwindigkeit von der Prozessorleistung des Rechners abhängig. Der Maya Hardware Renderer arbeitet mit Echtzeit und mit Hilfe der Grafikkarte. Der Vector Renderer ermöglicht es, vektorbasierte Bilder zu erstellen und kann als Zeichentrick Renderer für ein beliebiges Bildformat genutzt werden. [19]

Maya arbeitet mit einem rechtshändischem Koordinatensystem, wie in Abbildung 25 dargestellt.

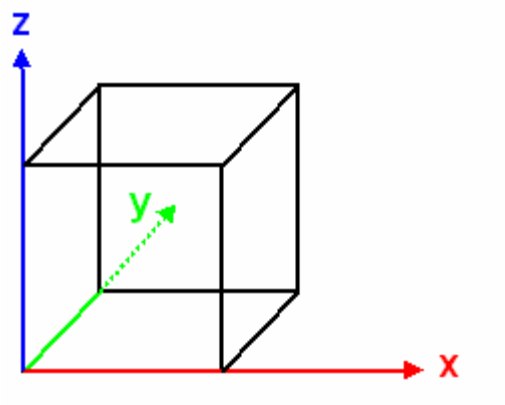


Abbildung 25: Koordinatensystem Maya

3.2 3Designer

3Designer ist das zentrale Softwarepaket zur Erstellung von zwei- und dreidimensionalen Grafiken und Animationen. Nachdem die virtuellen Szenenbilder in Maya erstellt und als VRML[5] exportiert wurden, können sie im 3Designer importiert werden und sind so im ORAD-System verarbeit- und darstellbar. Über den Grafikserver und eines der beiden Steuerungstools für den virtuellen Operator (3DPlay oder Maestro) werden die virtuellen Kulissen an die Renderengine (HDVG) weitergegeben und dort gerendert.

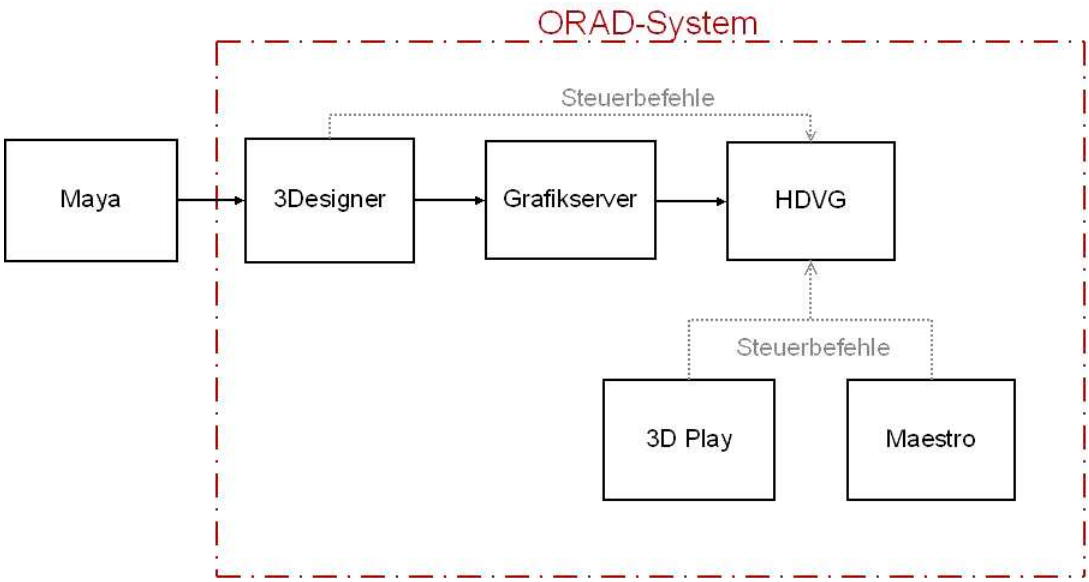


Abbildung 26: ORAD-System

[5] Die Virtual Reality Modeling Language (VRML, gesprochen Wörml) ist eine Skriptsprache, bzw. eine HTML-Erweiterung, die es erlaubt dreidimensionale Szenen darzustellen.

Es wird layerbasierend gearbeitet. Das bedeutet, dass Elemente, die den gleichen thematischen Inhalt haben, in einem Layer zusammengefasst werden können und diesem ein Licht und eine Kamera zugeordnet werden kann. Ein Layer kann mehrere Gruppen beinhalten. So können z.B. Objekte, die für ein bestimmtes Dekorationsteil im Set bestimmt sind, zu einer Gruppe zusammengefasst werden. Den einzelnen Objekten können Attribute wie Texturen oder Multitexturen, Farben oder Shader zugeordnet werden. Für jedes Attribut ist eine Spalte vorgesehen, in der dann das jeweilige Symbol erscheint.

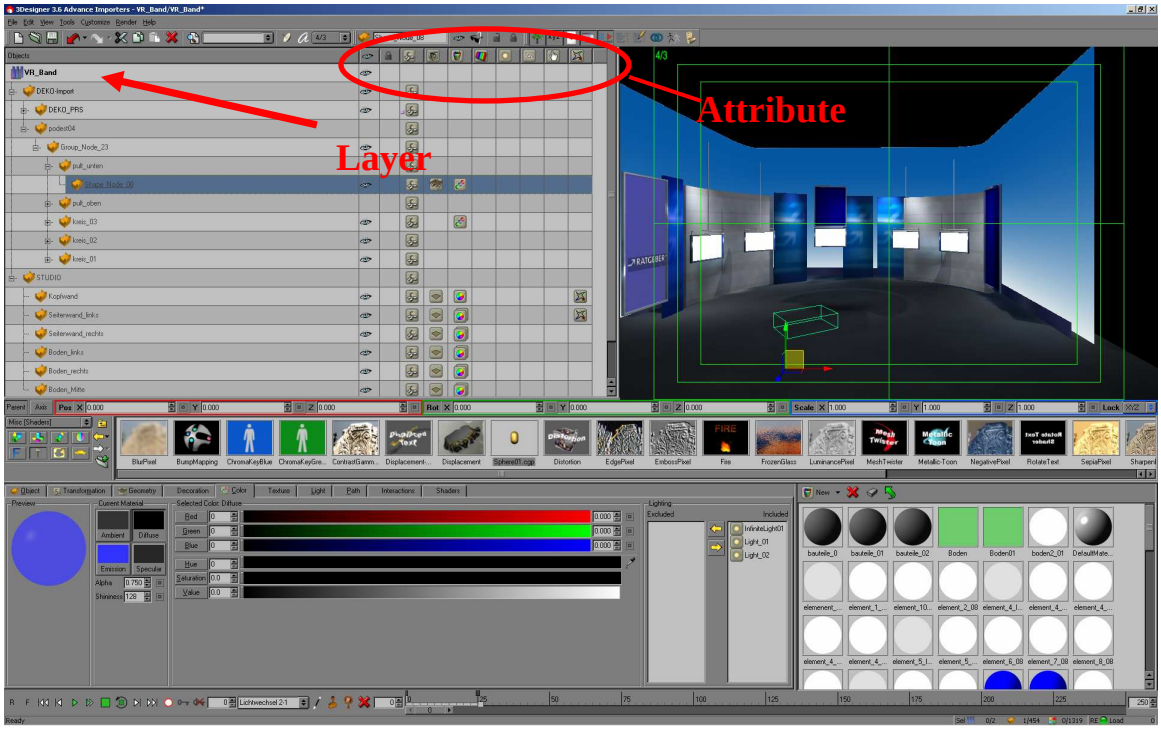


Abbildung 27: 3Designer

Des Weiteren können diverse Variablen, die mit dem User Interface zu beeinflussen sind, aus 3Designer exportiert werden. Dieser Vorgang ist für die Arbeit in den virtuellen Produktionen sinnvoll. Sind Exporte im 3Designer erstellt, so kann ein Operator von den Bediensoftwares *Maestro* oder *3D Play* auf diese Exporte zugreifen und die Variablen von dort aus ändern. Exportiert werden können Variablen, wie Objektpositionen, Farben, Texturen und viele andere. Im WDR Köln wird besonders mit der Objektposition als Export gearbeitet. So können einzelne Teile des Sets (z.B. ein Monitor) an die Gegebenheiten, wie die Größe eines Moderators oder einer Moderatorin, angepasst werden, indem die Objektposition in der Höhe verschoben wird.

Alle Variablen, die exportierbar sind, können auch über so genannte Connections mit anderen Variablen verbunden werden. Hierbei ist es auch möglich, eine mathematische Funktion zwischen den Variablen zu erstellen. Eine Objektfarbe könnte so zum Beispiel abhängig gemacht werden von der Objektposition. [20]

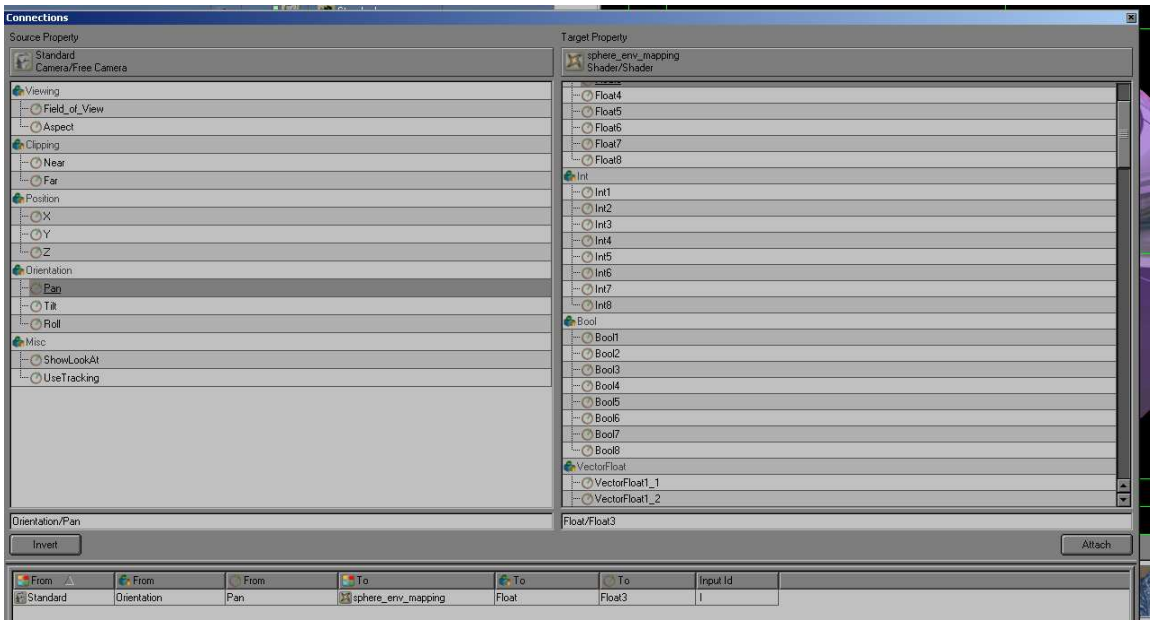


Abbildung 28: Connections

Im 3Designer wird mit einem Koordinatensystem gearbeitet, dessen Achsen anders stehen, als die in Maya. In Abbildung 29 sind die Unterschiede der Koordinatensysteme zu erkennen.

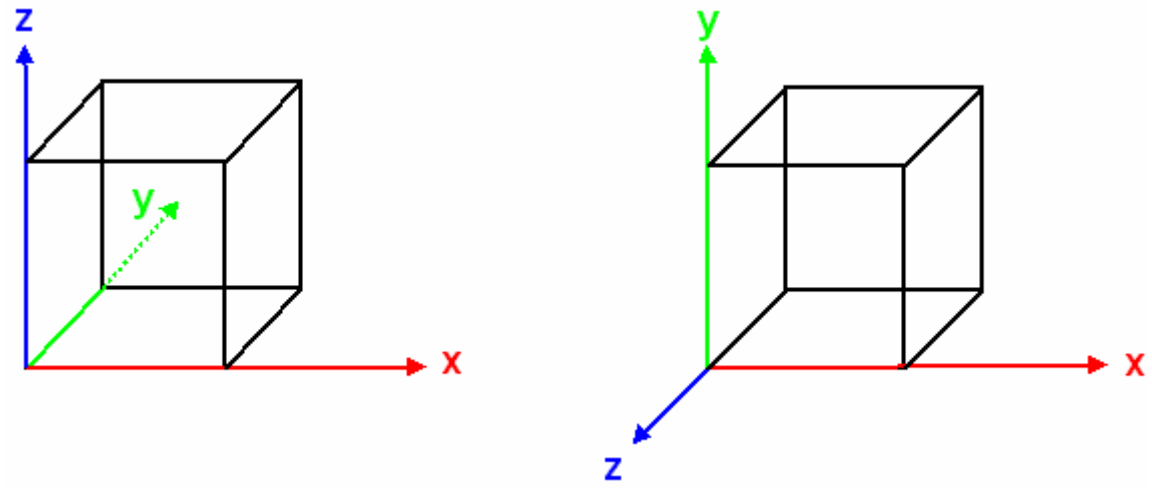


Abbildung 29: Vergleich Koordinatensysteme Maya (links) und 3Designer (rechts)

3.3 Koordinatensysteme und Transformationsmatrizen in Maya und 3Designer

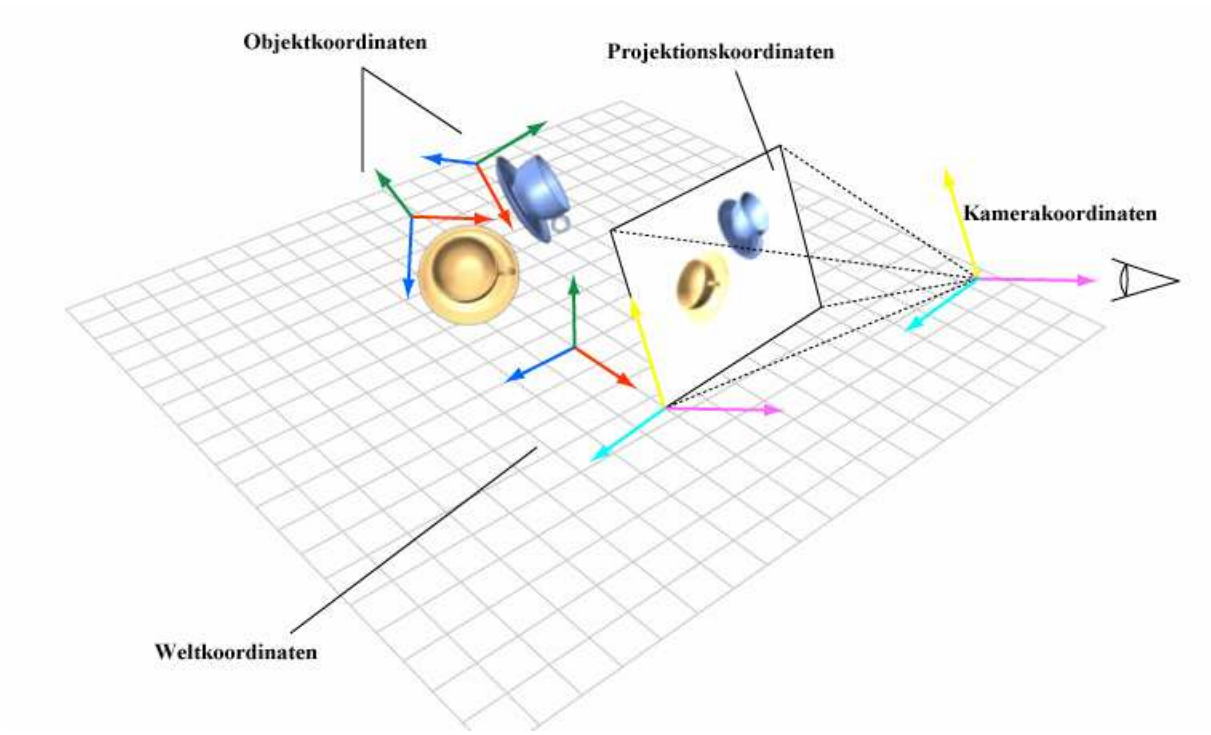


Abbildung 30: Koordinatensysteme [21]

In Abbildung 30 sind alle Koordinatensysteme zusammen gefasst, die ein Punkt auf der Oberfläche eines Objektes in der Pipeline durchlaufen muss. Das Objekt wird im Objektkoordinatensystem erstellt. Das Objekt besitzt eine Position im Raum, die in Weltkoordinaten angegeben werden kann, eine Position, die von der Kamera aus angegeben werden kann (mit der Kamera im Ursprung des Koordinatensystems) und eine auf der zweidimensionalen Projektionsebene. Um die Koordinaten eines Objektes vom Objektkoordinatensystem in das Weltkoordinatensystem umzurechnen wird eine Transformationsmatrix benötigt. In dieser Matrix stecken die Informationen, in welcher Position sich das Objekt in Weltkoordinaten befindet (Formel 11), wie das Objektkoordinatensystem im Bezug zum Weltkoordinatensystem gedreht ist (Formel 12) und ob es anders skaliert wurde (Formel 13) als das Weltkoordinatensystem. Multipliziert man diese Matrizen miteinander, welche die jeweiligen Informationen beinhalten, so erhält man eine Matrix, die zur Umrechnung vom Objekt- in das Weltkoordinatensystem dient.

$$T(x,y,z)=\begin{pmatrix}1&0&0&x\\0&1&0&y\\0&0&1&z\\0&0&0&1\end{pmatrix}$$

(Formel 11: Translationsmatrix)

$$R_x(\alpha)=\begin{pmatrix}1&0&0&1\\0&\cos\alpha&-\sin\alpha&0\\0&\sin\alpha&\cos\alpha&0\\0&0&0&1\end{pmatrix}$$

$$R_y(\alpha)=\begin{pmatrix}\cos\alpha&0&\sin\alpha&1\\0&1&0&0\\-\sin\alpha&0&\cos\alpha&0\\0&0&0&1\end{pmatrix}$$

$$R_z(\alpha)=\begin{pmatrix}\cos\alpha&-\sin\alpha&0&0\\\sin\alpha&\cos\alpha&0&0\\0&0&1&0\\0&0&0&1\end{pmatrix}$$

(Formel 12: Rotationsmatrizen)

$$S(x,y,z)=\begin{pmatrix}x&0&0&0\\0&y&0&0\\0&0&z&0\\0&0&0&1\end{pmatrix}$$

(Formel 13: Skalierungsmatrix)

Um vom Weltkoordinatensystem ins Kamerakoordinatensystem zu kommen, benötigt man die Position der Kamera in Weltkoordinaten. Diese wird in die Formel 11 eingesetzt. Auch die Rotation und Skalierung werden mit einberechnet und es entsteht eine zweite Matrix: die Umrechnung von Welt- in Kamerakoordinaten. Um von Objekt- in Kamerakoordinaten umzurechnen, müssen die beiden Matrizen miteinander multipliziert werden (wobei zu beachten ist, dass Matrixmultiplikationen nicht kommutativ sind, für zwei Matrizen A und B gilt: $A \cdot B \neq B \cdot A$).

Am Ende der Pipeline werden die Daten am Bildschirm ausgegeben. Dafür muss die Berechnung von dreidimensionalen Kamerakoordinaten in zweidimensionale Projektionskoordinaten durchgeführt werden.

Bei der Arbeit mit Shadern hat man die Möglichkeit auf diese Matrizen zu zugreifen.

In CgFX kann, wie in Kapitel 2.6.2 erläutert, über die Semantics auf Werte der Softwareumgebung zugegriffen werden.

Für die Transformationsmatrizen gibt es zahlreiche Möglichkeiten:

WORLD	WORLDVIEWPROJECTIONINVERSE
VIEW	WORLDTRANSPOSE
PROJECTION	VIEWTRANSPOSE
WORLDVIEW	PROJECTIONTRANSPOSE
VIEW PROJECTION	WORLDVIEWTRANSPOSE
WORLDVIEWPROJECTION	VIEW PROJECTIONTRANSPOSE
WORLDINVERSE	WORLDVIEWPROJECTIONTRANSPOSE
VIEWINVERSE	WORLDINVERSETRANSPOSE
PROJECTIONINVERSE	VIEWINVERSETRANSPOSE
WORLDVIEWINVERSE	PROJECTIONINVERSETRANSPOSE
VIEW PROJECTIONINVERSE	WORLDVIEWINVERSETRANSPOSE
	VIEW PROJECTIONINVERSETRANSPOSE
	WORLDVIEWPROJECTIONINVERSETRANSPOSE

Abbildung 31: Semantics – Transformationsmatrizen [18, Seite 33f]

WORLD ist die Matrix zur Umrechnung von Objekt- in Weltkoordinaten, VIEW die Umrechnung von Welt- in Kamerakoordinaten und PROJECTION von Kamera in Projektionskoordinaten. WORLDVIEW ist die Matrix, die als Ergebnis der Multiplikation von der VIEW- und der WORLD-Matrix vorliegt. So kann man auch auf alle kombinierten Matrizen zugreifen. Die Anhänge INVERSE und TRANSPOSE geben die Inverse oder Transponierte der jeweiligen Matrix aus. In CgFX-Shadern ist es daher kein Problem, in den verschiedenen Koordinatensystemen Berechnungen durchzuführen.

Anders sieht es in den Cg-Shadern für den 3Designer aus. Da mit dem ARB-Profil (für Vertexshader arbvpl und für Fragmentshader arbfpl) gearbeitet wird, gibt es die Möglichkeiten auf OpenGL-States[6] zu zugreifen. In dem Nutzerhandbuch des Cg-Toolkits von NVIDIA sind folgende OpenGL-States für Matrizen aufgelistet, auf die man mit dem Profil Zugriff hat:

[6] OpenGL arbeitet mit Zustandsautomaten. Dies ist ein Model der Informationsverarbeitung, das auf internen Zuständen des Systems basiert. Das bedeutet, dass Eingaben, abhängig von ihrem derzeitigen Zustand, in den Speicher geschrieben werden. Der interne Zustand des OpenGL-Automaten wird auch als Status bezeichnet. [22, Seite 14]

state.matrix.modelview[0]	state.matrix.projection
state.matrix.mvp	state.matrix.texture[0]
state.matrix.palette[0]	state.matrix.program[0]
state.matrix.inverse.modelview[0]	state.matrix.inverse.projection
state.matrix.inverse.mvp	state.matrix.inverse.texture[0]
state.matrix.inverse.palette[0]	state.matrix.inverse.program[0]
state.matrix.transpose.modelview[0]	state.matrix.transpose.projection
state.matrix.transpose.mvp	state.matrix.transpose.texture[0]
state.matrix.transpose.palette[0]	state.matrix.transpose.program[0]
state.matrix.invtrans.modelview[0]	state.matrix.invtrans.projection
state.matrix.invtrans.mvp	state.matrix.invtrans.texture[0]
state.matrix.invtrans.palette[0]	state.matrix.invtrans.program[0]

Abbildung 32: OpenGL – States [15, Seite 257]

Hier erkennt man die modelview-Matrix. In CgFX ist dies die WORLDVIEW-Matrix, also die Umrechnung von Objektkoordinaten in Kamerakoordinaten. Auf die einzelnen Matrizen zur Umrechnung von Objekt- in Weltkoordinaten oder von Welt- in Kamerakoordinaten hat man keinen Zugriff. Es gibt also keine Möglichkeit in Weltkoordinaten zu arbeiten, da die Matrix zur Umrechnung in den Weltkoordinatenraum nicht abrufbar ist.

Die mvp-Matrix bedeutet „Model-View-Projection“-Matrix und rechnet von Objekt- in Projektionskoordinaten um. Auch hier gibt es inverse und transponierte Matrizen (invtrans ist die Kombination aus Inverser und Transponierter der Matrix). [18]

In Abbildung 33 sind einige Umrechungen zwischen den Koordinatensystemen aufgezeigt.

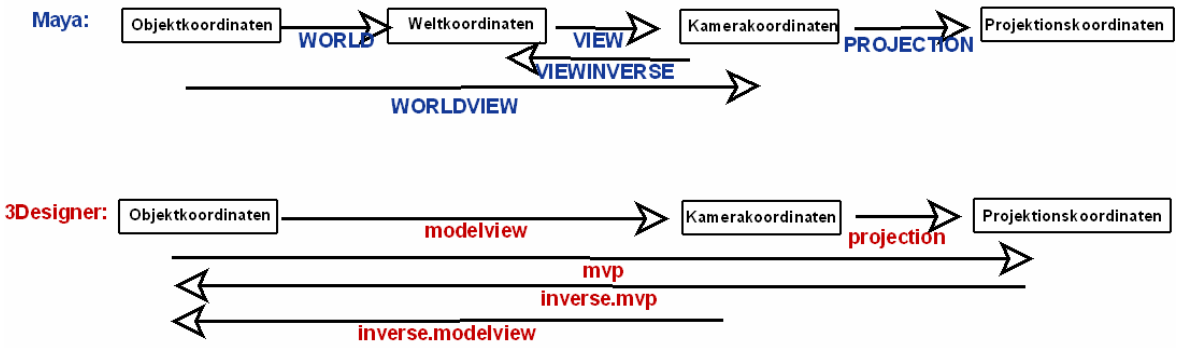


Abbildung 33: Transformationsmatrizen in Maya und 3Designer

Normalenvektoren verhalten sich bei Koordinatentransformationen nicht wie die Vertices, sondern wie die Flächen, auf denen sie senkrecht stehen. Deshalb müssen sie mit der Invers Transponierten der jeweiligen Matrix multipliziert werden, um in ein anderes Koordinatensystem transformiert zu werden. [5, Seite 205]

4 Einbindung von Shadern

Nachfolgend wird erläutert, wie in die Softwareprogramme Maya und 3Designer ein Shader eingebunden wird.

4.1 Maya

In Maya muss erst ein Plugin (Plugins erweitern die Funktionalität von Softwareprogrammen) aktiviert werden um mit CgFX-Shadern zu arbeiten. Unter „Window“ → „Settings/Preferences“ → „Plug-in Manager“ muss hierfür bei dem CgFX-Plugin ein Haken gesetzt werden. Anschließend kann man im Hypershade unter den Render-Nodes einen neuen CgFX-Shader erstellen. Im Attribute Editor erscheint dann ein CgFX-Shader-Material-Knoten.

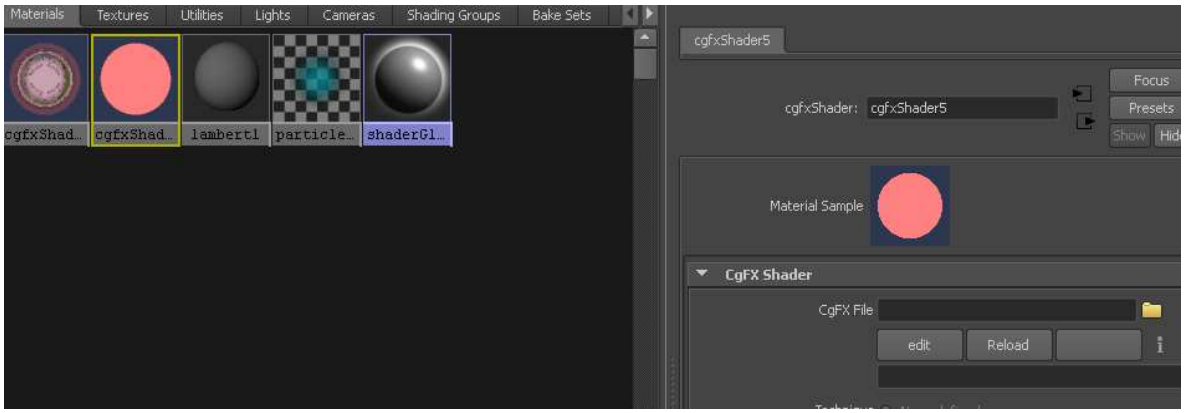


Abbildung 34: Shader einbinden

Unter CgFX-File wird anschließend der Pfad zum File ausgewählt. Sind im Shader Annotations vorhanden, so erscheint in Maya automatisch mit dem Laden des Shaders ein User Interface. Der Shader kann auf ein Objekt angewandt werden, indem man dieses dem Shader über den Befehl „assing to material“ zuordnet. [23]

4.2 3Designer

Um einen Shader im 3Designer einzubinden, muss eine Cg-Datei für den Shader in den Ordner „Shaders“ abgelegt werden. Dies kann sowohl ein Vertex, als auch ein Fragmentshader sein. Sollen in einem Shader beide benutzt werden, so muss für jeden ein File in den Ordner abgelegt werden.

Anschließend gibt es zwei Möglichkeiten den Shader einzubinden.

Mit Rechtsklick auf den Shader-Container-Reiter kann ein neuer Shader-Container erstellt werden (Abbildung 35). Über „New“ muss anschließend ein neuer Shader erstellt werden (Abbildung 36).

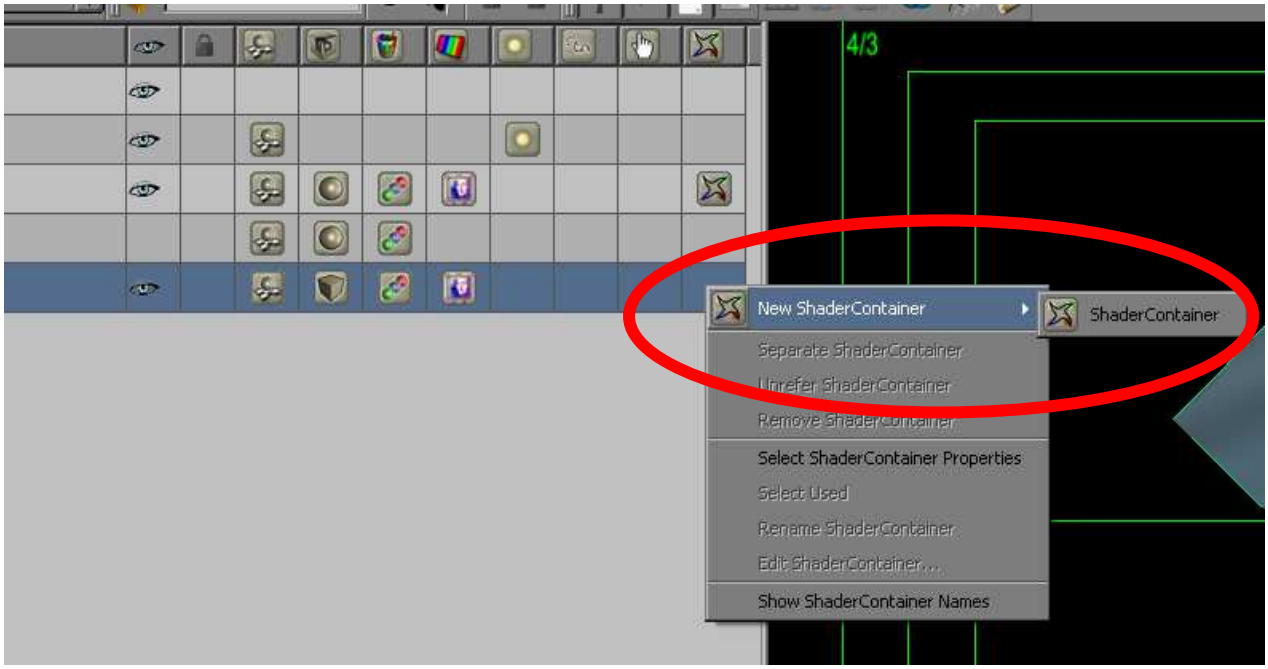


Abbildung 35: neuen Shader-Container erstellen

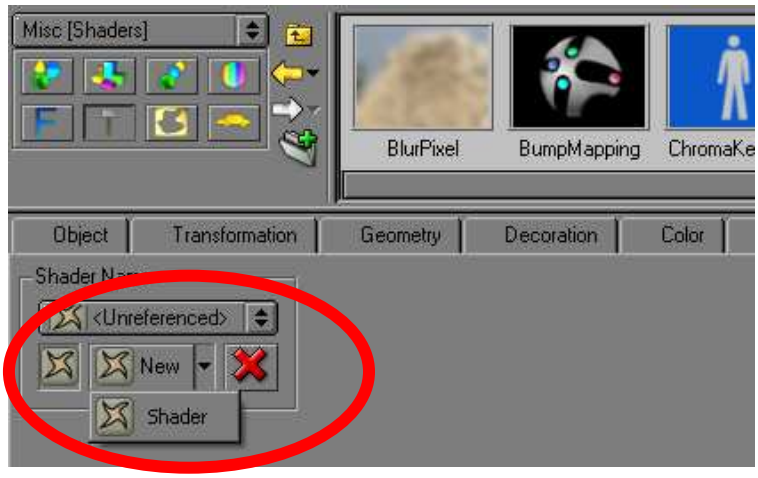


Abbildung 36: neuer Shader

Nun muss die Verbindung zu den Cg-Dateien hergestellt werden. Im Advanced Editor im PropertiesView unter „Other“ findet man unter „Shader“ alle in der Szene eingebundenen Shader. Im Property Editor können anschließend in den Vertex- und Fragmentshaderdatenfeldern XML-Anweisungen eingegeben werden, die die Verknüpfung zum Shader beinhalten (vgl. Abbildung 37).

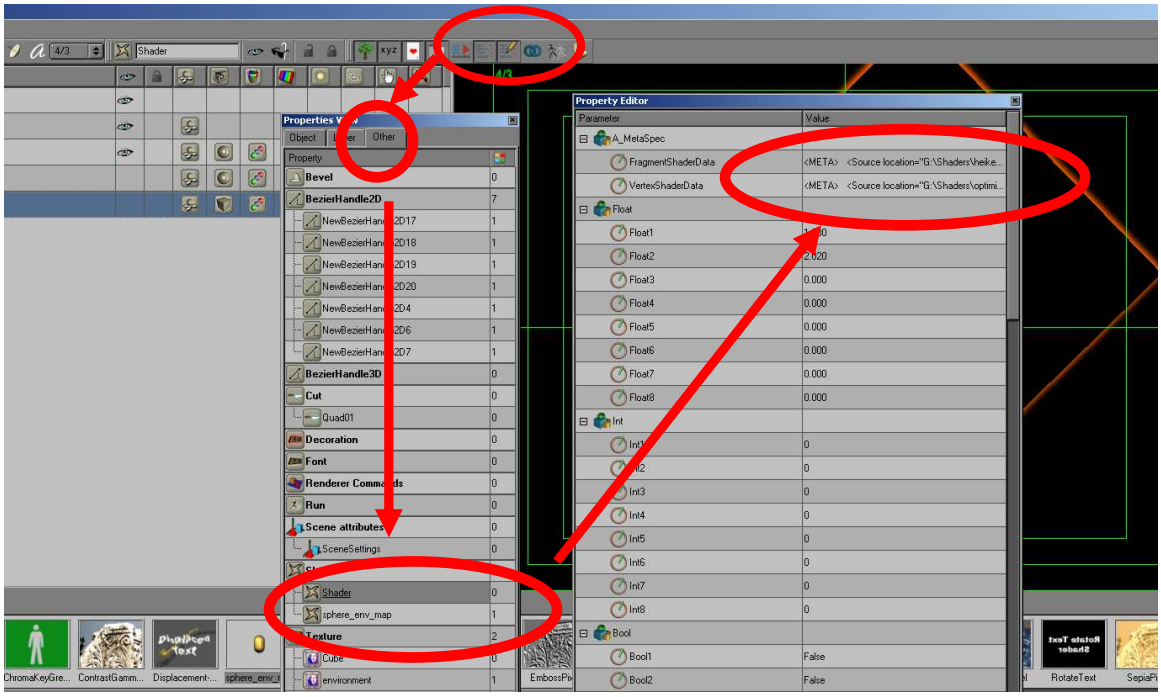


Abbildung 37: Eingabefelder für XML-Anweisung

So eine XML-Datei setzt sich aus zwei Teilen zusammen. Der erste Teil verknüpft das User Interface mit der Cg-Datei, der zweite Teil erstellt auf dem User Interface Bedienelemente für den Shader. Abbildung 38 zeigt ein Beispiel.

```
<META>
      <Source location=“G\Shaders\TwirlTextureFragmentShader.cg” />
      <CG>
        <params>
         <Parameter name= “twirl_rotation” type= “float”>
          <Mapping name = “Float1”/>
         </Parameter>
        </params>
      </CG>
      <DGUI caption= “Twirl texture”>
       <Ctrl type= “double_spin_box” label= “Level:” group_name= “Float”
        param_name= “Float1” row= “0” col= “0” max_width= “50”
        height= “”/>
      </DGUI>
</META>
```

Abbildung 38: XML-Anweisung

Unter „source location“ wird der Pfad zur Datei angegeben. Der Inhalt zwischen <params> und </params> beinhaltet die Verknüpfung zu einem Parameter im Shadercode. Hier können beliebig viele eingesetzt werden. Der Parametername und -typ müssen identisch sein mit dem im Programmcode genutzten Parameter (hier float twirl_rotation). Im Shadercode ist das Schlüsselwort uniform, so wie in Kapitel 4.2 beschrieben, zu nutzen. Der „Mapping name“ ist der Parametername, der im zweiten Teil der Anweisung einem Bedienelement zugeordnet wird.
Der zweite Teil beginnt mit dem Namen des Bedienelements, welcher hinter „DGUI caption“ einzufügen ist. Zwischen „<Ctrl“ und „/>“ stehen die Befehle für ein Bedienelement. Hier können wieder beliebig viele eingefügt werden.
„type“ beschreibt den Typ des Bedienelements. Dazu gehören: „spin_box“ für Intwerte, „double_spin_box“ für Floatwerte, „combo_box“ für State-Parameter in Vertex-Shadern und „texture_reference“ für Texturparameter in Fragment-Shadern. Hinter „label“ steht der Name des Elements, so wie er auf dem User Interface stehen soll. Der „group_name“ gibt noch einmal den Typ des Parameters an und der „param_name“ muss genau den Angaben des „Mapping_name“ im ersten Teil entsprechen. Das Bedienelement kann des Weiteren mit „col“ und „row“ platziert, und mit „max_width“, „height“, „columns“ in der Größe angepasst werden. „min_value“, „max_value“, „precision“ und „step“ stellen den Wertebereich ein (nur gültig für „double_spin_box“). Ein „texture_reference“-Element kann noch über „LeftToRight“, „RightToLeft“, „BottomToTop“ oder „TopToBottom“ mit dem Befehl „layout“ angeordnet werden.

Der zweite Weg, einen Shader einzubinden, ist über den Menüpunkt „Tools“ unter „Shader“. Dort öffnet sich ein Dialogfenster, wo man einen Pfad zum Vertex- und einen zum Fragmentshader herstellen kann (vgl. Abbildung 39).

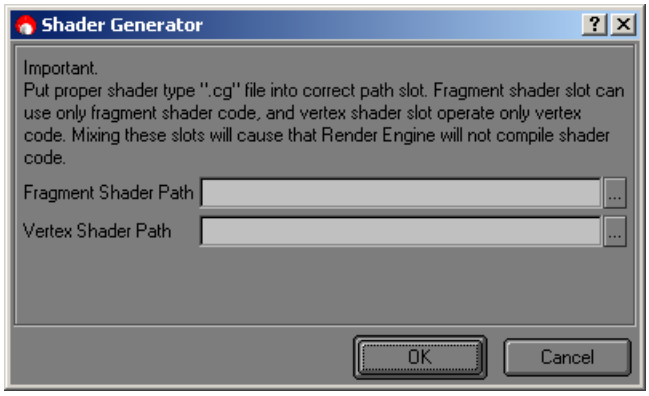


Abbildung 39: Shader einbinden

Dafür muss das Objekt, welches mit dem Shader versehen werden soll, vorher markiert sein. 3Designer produziert bei diesem Vorgang ein User-Interface, welches von den uniform-Parametern im Shader abhängt. Allerdings sind die Bedienelemente dann willkürlich angeordnet und es werden auch nicht immer alle Parameter übernommen. Deshalb sollten hier, wie in der anderen Vorgehensweise, die XML-Anweisungen über den in Abbildung 37 gezeigten Weg eingefügt werden. Funktionierende Shadercontainer können per drag/drop in die Shaderlibrary aufgenommen werden. [24]

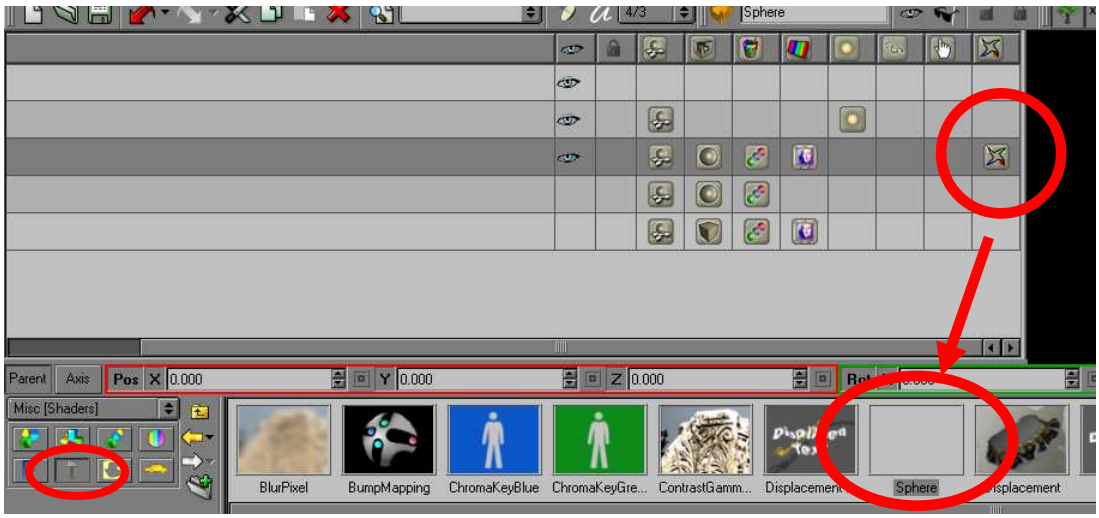


Abbildung 40: Shaderlibrary

5 Workflow zur Erstellung virtueller Szenenbilder im WDR Köln

In diesem Kapitel wird der Workflow zur Erstellung eines virtuellen Szenenbildes im WDR Köln an dem Beispiel des Szenenbildes zu der Sendung „Ratgeber Recht“ demonstriert.

In Maya werden die virtuellen Kulissen wie in Abbildung 41 erstellt.

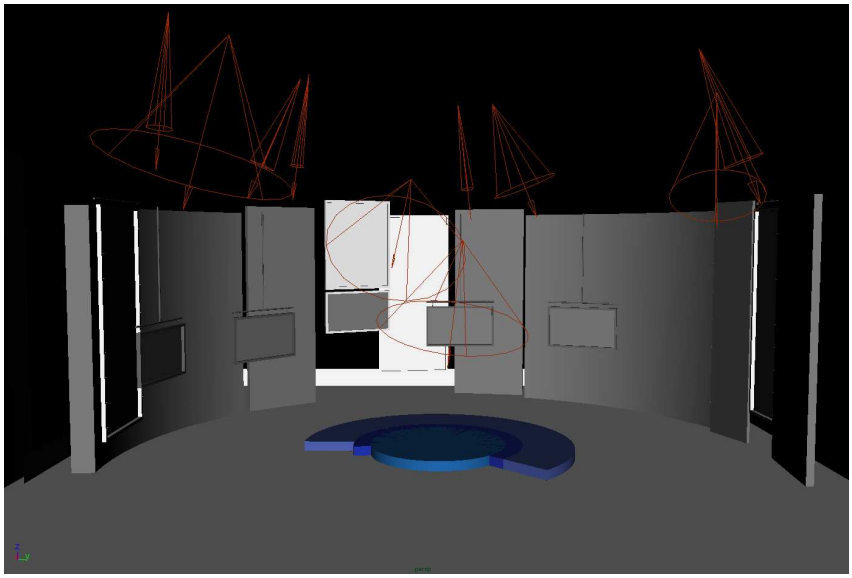


Abbildung 41: Workflow01

Anschließend werden sie mit Texturen belegt (vgl. Abbildung 42). Für die Darstellung wird das Hardware-Rendering genutzt. Die Beleuchtung wird lokal über Lambert-Shader realisiert.

Abbildung 42: Workflow02

Um eine bessere Qualität der Darstellung zu erreichen wird in Maya ein Software-Rendering vollzogen (vgl. Abbildung 43).

Abbildung 43: Workflow03

Man sieht Lichter und Schatten auf den Oberflächen der Objekte. Für jedes Objekt wird die Oberfläche inkl. Lichter und Schatten als Textur gespeichert. Abbildung 44 zeigt die hintere rechte Wand, auf welcher der Schatten des Monitors und die Lichter erkennbar sind.

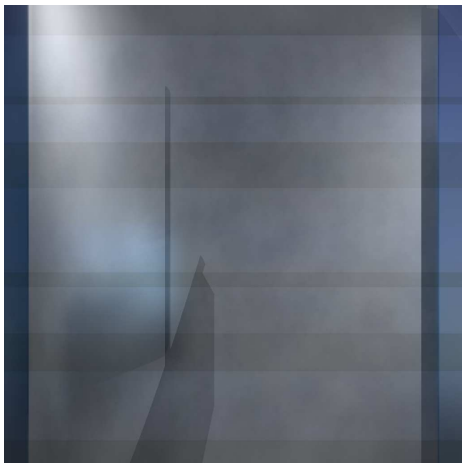


Abbildung 44: Textur

Diese Texturen werden anschließend auf die Objekte aus Abbildung 41 gelegt. So wurde ein qualitativ hochwertiges Modell erstellt, welches echtzeitfähig ist. Lichter und Schatten sind mit einberechnet, Reflexionen der Umgebung fehlen allerdings, was dem Szenenbild, sobald sich eine Kamera darin bewegt, ein starres Aussehen verleiht. Um dies zu ändern, wird der Environment-Shader benötigt. Zur Erstellung der Environment-Textur wird in den

Mittelpunkt des Sets eine Kugel gelegt mit der optimal spiegelnden Materialeigenschaft von Chrom (vgl. Abbildung 43). Die Spiegelung der gesamten Umgebung auf der Kugel wird als Textur abgespeichert (vgl. Abbildung 45).
Da die Seite der Kugel, die zur virtuellen Kamera zeigt, schwarz darstellen würde, wurde ein Foto vom Studio erstellt und in diesen Bereich gelegt.

Abbildung 45: Textur für die Umgebungsreflexion

In Abbildung 46 wurde ein Objekt in das Set gestellt, welches mit dem Shader belegt werden soll. Vor dem Export des Szenenbildes wäre es für den Designer sinnvoll, hier den Environment-Shader in Maya zur Verfügung zu haben. So könnte die Gestaltung des Objektes (zum Beispiel zur Bestimmung der Kanten, ob sie abgerundet oder scharfkantig mit Reflexionen besser aussehen) mit Hilfe des Shaders optimiert werden.

Abbildung 46: Workflow04

Als Ausgangssituation liegt in Maya zu diesem Zeitpunkt also das Set aus dreidimensionalen Objekten, die Texturen (die Schatten und Lichter beinhalten) und eine zweidimensionale Textur (die für die Umgebungsreflexion vorbereitet ist) vor. Das Set wird über das Format VRML aus Maya exportiert und im 3Designer importiert (vgl. Abbildung 47).

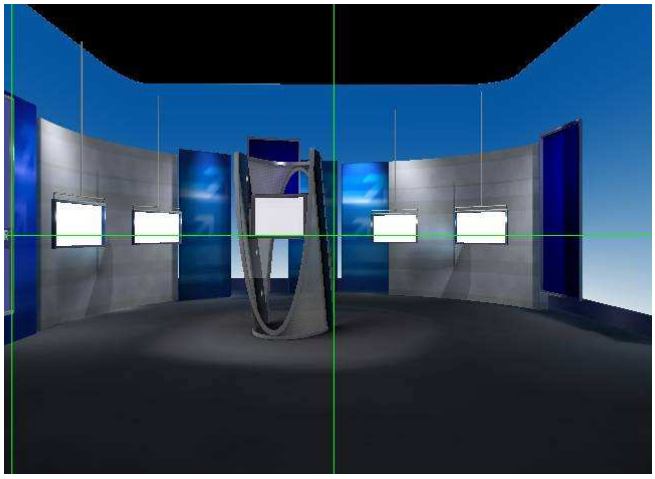


Abbildung 47: Workflow05

Die Monitore sind hier weiß dargestellt, als Input dient später ein Videosignal. Abbildung 48 zeigt einen Ausschnitt des Objektes bei dem auf den Holzstreben nur die Textur aus Maya gelegt wurde. Abbildung 49 zeigt die Anwendung des in dieser Arbeit entstandenen Environment-Shaders. Hier ist das Holz zu einem hoch glänzend lackierten Holz geworden, welches zusätzlich zu seiner Holzeigenschaft die Umgebung reflektieren kann.

Abbildung 48: Workflow06

Abbildung 49: Workflow07

Abbildung 50: Darstellung des Szenenbildes im Endergebnis

Abbildung 50 zeigt wie das Szenenbild genutzt wird. Der Tisch ist die einzige reale Dekoration in diesem Set. Die Moderatorin und der Tisch stehen im Studio E des WDR Köln in einem grünen Raum.

6 Bedingungen an den Shader

Betrachtet man den Workflow, so ist vor allem wichtig, dass die Umgebungstextur auf einer Kugel abgebildet werden kann, da die Erstellung der Textur mit der Chromkugel problemlos umsetzbar ist. Das Sphere-Environment-Mapping aus Kapitel 2.4.3 ist nicht geeignet, da die Reflexionen nur für einen Blickwinkel gelten. Das Cubic-Environment-Mapping ist blickwinkelunabhängig, allerdings ist die Herstellung der Textur sehr aufwändig.

Weitere Ansprüche an den Shader ergaben sich durch den Wunsch, die Umgebungstextur flexibel gestalten zu können. Sie sollte drehbar, skalierbar und verschiebbar sein. Ihre Intensität sollte einstellbar sein, da Objekte unterschiedlich stark reflektieren.

Des Weiteren sollte die Möglichkeit bestehen die Reflexionen mit einer Textur, die auf dem Objekt abgebildet ist, zu mischen.

Auf Abbildung 27 ist zu erkennen, dass es verschiedene Reiter gibt, die jeweils über ihr eigenes Bedienfeld einstellbar sind. So gibt es für jedes Objekt z.B. für Texturen, Lichter und Farben ein eigenes Bedienfeld. Eine weitere Anforderung an den Shader in 3Designer bestand darin, dass diese Bedienfelder weiter genutzt werden können. So soll eine Textur über ihr eigenes Bedienfeld eingefügt und verändert werden können und die gesetzten Parameter in den Shader mit eingebunden werden. Auch eine Objektfarbe oder Lichter, die in der Szene gesetzt wurden, sollen mit Verwendung des Shaders nicht ignoriert werden.

Von geringer Bedeutung sind die Lichtberechnungen auf den Objekten in Maya. Denn in einem Szenenbild in Maya, werden keine Lichter mehr gesetzt, sobald ein Software-Rendering durchgeführt wurde. Sollten weitere Lichter gesetzt werden, geschieht dies im 3Designer, worin die Lichtberechnung weiter einbezogen werden sollte, wie vorher beschrieben.

7 Vorstellung der Shader

In diesem Kapitel werden die Shader, welche in der vorliegenden Arbeit erstellt wurden, detailliert vorgestellt.

7.1 CgFX für Maya

Am Anfang des CgFX-Shaders werden die Transformationsmatrizen über die Semantics in den Shader geladen. Es werden die Matrizen zur Umrechnung von Objekt- in Projektionskoordinaten und von Objekt- in Weltkoordinaten benötigt. Für die Normalenvektoren wird die Invers Transponierte der Transformationsmatrix genutzt, die von Objekt- in Weltkoordinaten umrechnet und zuletzt die Transformationsmatrix, die die Transformation vom Kamerakoordinatensystem zurück ins Weltkoordinatensystem beschreibt.

```
float4x4 worldviewprojection        : WORLDVIEWPROJECTION;
float4x4 world                      : WORLD;
float4x4 worldinversetranspose      : WORLDINVERSETRANSPOSE;
float4x4 viewinverse                : VIEWINVERSE;
```

Listing 1: Transformationsmatrizen

Anschließend folgen die Annotations. Hier gibt es zwei Texture-Samplers, eine für die Textur, die als Umgebungstextur genutzt werden soll, und eine, die auf das Objekt abgebildet werden kann. Acht verschiedene Schieberegler werden mit dem UIWidget „Slider“ versehen. „transform U“, „transform V“, „rotate“, „scaleU“ und „scaleV“ beeinflussen die Umgebungstextur. Diese kann so beliebig verschoben, skaliert oder gedreht werden, gerade so, wie sie auf dem Objekt erscheinen soll. Der Parameter „intensity Environment“ gibt im Bereich zwischen Null und Eins an, mit wie viel Prozent die Umgebung reflektiert werden soll. „intensity Surface“ bestimmt die Intensität der Textur auf der Oberfläche. Die bool-Variable „TextureMix“ schaltet die Mischung der beiden Texturen ein. Wird dieses Häkchen nicht gesetzt, so ist auf dem Objekt nur die reflektierte Umgebungstextur zu sehen. Das User Interface sieht in Maya damit so aus, wie in Abbildung 51.

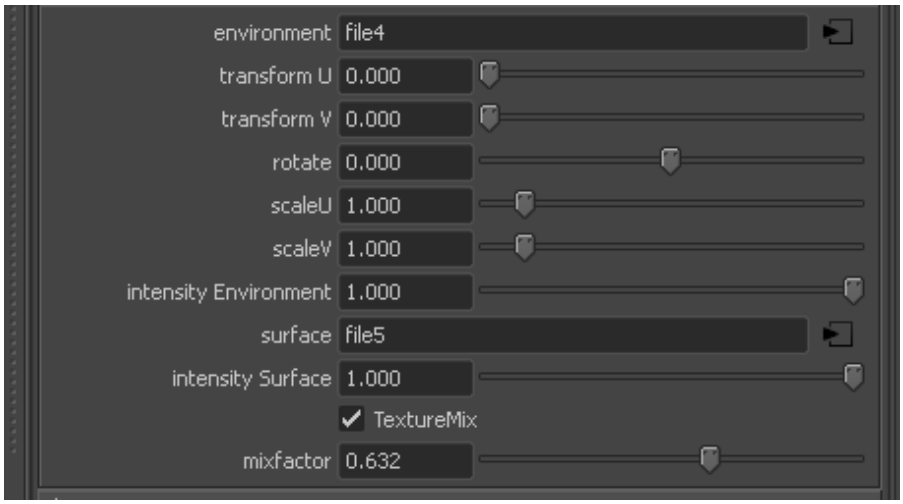


Abbildung 51: User Interface in Maya

Nach den Annotations folgen zwei Strukturen. (Strukturen fassen verschiedene Komponenten zusammen, die unterschiedliche Datentypen haben können [25].) Die Struktur vertexInput beinhaltet drei Parameter, die von der Pipeline in den Vertex-Shader gegeben werden. *Position* ist die Position des Vertex in Objektkoordinaten des jeweiligen zugehörigen Objektes, *UV* sind Texturkoordinaten, die diesem Vertex zugeordnet werden. *Normal* ist der Normalenvektor des Vertex ebenfalls in Objektkoordinaten.

```
struct vertexInput
{     float4 Position   : POSITION0;          //Objektkoordinaten
      float4 UV         : TEXCOORD0;          //Texturkoordinaten
      float3 Normal     : NORMAL;             //Objektkoordinaten
};
```

Listing 2: Struktur vertexInput

In der Struktur vertexOutput wird definiert, welche Parameter vom Vertex-Shader wieder an die Pipeline übergeben werden. Gleichzeitig sind dies auch die Werte, die interpoliert von der Pipeline dem Fragmentshader übergeben werden. Wobei *Position* die Vertexposition in Projektionskoordinaten ist und immer an die Pipeline übergeben werden muss, da sie fürs Clipping benötigt wird. *PosO* ist die Position in Objektkoordinaten, *UV* sind die Texturkoordinaten und *Normal* der Normalenvektor in Weltkoordinaten. Als Semantics werden die POSITION und die TEXCOORD0-2 genutzt.

```
struct vertexOutput
{     float4 Position   : POSITION0;          //Projektionskoordinaten
      float2 UV         : TEXCOORD0;          //Texturkoordinaten
      float4 PosO       : TEXCOORD1;          //Objektkoordinaten
      float3 Normal     : TEXCOORD2;          //Weltkoordinaten
};
```

Listing 3: Struktur vertexOutput

Die Umrechnungen der Parameter in andere Koordinatensysteme erfolgen im Vertex-Shader. Mit der Matrix *worldviewprojection* wird die Position in Projektionskoordinaten und mit *worldinversetranspose* die Normale in Weltkoordinaten umgerechnet.
PosO gibt die Position in Objektkoordinaten weiter, *UV* gibt die x- und y-Komponente der Texturkoordinaten als Vektor weiter. Dies ist zwar kein Vektor, aber so besteht eine einfache Möglichkeit auf diese Werte zu zugreifen. Als Output-Möglichkeit wurde hier die Variante gewählt, bei dem ein Output-Container erstellt wird und dieser, mit Werten gefüllt, wieder ausgegeben wird.

```
vertexOutput VP(vertexInput Input)
{
      vertexOutput Output;

      Output.Position = mul(worldviewprojection, Input.Position);
      Output.PosO = Input.Position;
      float3 Normal = normalize(mul((float3x3)worldinversetranspose,
                                                      Input.Normal));
      Output.Normal = Normal;
      Output.UV = Input.UV.xy;
      return Output;
}
```

Listing 4: Vertex-Shader

Der Fragment-Shader bekommt als Input den zuvor definierten Output des Vertex-Shaders. Zuerst wird in dem Shader die Koordinate auf der Umgebungstextur berechnet, welche auf dem Fragment, aus Sicht der Kamera, abgebildet wird. Dafür wird die Methode „calc_sphere_coords“ genutzt. Mit der in Cg vorhandenen Methode „tex2D()“ wird die Farbe des Pixels auf diesem Punkt in *sphereRGBA* gespeichert. In *surfaceRGBA* wird die Farbe gespeichert, welche die auf dem Objekt abgebildete Textur in dem Fragment beinhaltet. Anschließend werden die prozentualen Anteile der beiden Texturen berechnet, abhängig von den im User Interface angegebenen Intensitäten. Über die Methode „multitex()“ werden die beiden Farben für das Fragment verrechnet und anschließend über return und der Semantic COLOR ausgegeben.

```
float4 FP(vertexOutput Input) : COLOR {
      //Berechnung der Koordinate auf der Environmenttextur
      float2 texCoord = calc_sphere_coords(Input.PosO, Input.Normal);

      //Berechnung der Farbe auf der errechneten Koordinate
      float4 sphereRGBA = tex2D(environment, texCoord);

      //Berechnung der Farbe auf der festen Textur
      float4 surfaceRGBA = tex2D(surface, Input.UV);
```

```
        //Intensität der Environmenttextur
        sphereRGBA = float4 (   (sphereRGBA.r * intensityE),
                                (sphereRGBA.g * intensityE),
                                (sphereRGBA.b * intensityE),
                                sphereRGBA.a);

        //Intensität der festen Textur auf der Oberfläche
        surfaceRGBA = float4 (  (surfaceRGBA.r * intensityS),
                                (surfaceRGBA.g * intensityS),
                                (surfaceRGBA.b * intensityS),
                                surfaceRGBA.a);
        //Mischen der Texturen
        float4 mulTex = multitex(surfaceRGBA, sphereRGBA);
        return mulTex;
}
```

Listing 5: Fragment-Shader

Die wichtigste Methode ist die Methode „calc_sphere_coords“. Hier wird bestimmt, welche Koordinate in dem Fragment, von Kamerasicht aus, reflektiert wird und somit die gesamte Berechnung vollzogen, die diesen Environment-Shader ausmacht. Wie in Kapitel 2.4.3 beschrieben, muss der Reflexionsvektor (der auf die Umgebungstextur zeigt) berechnet werden. Dies ist der reflektierte Strahl von dem Vektor **U**, der von der Kamera auf das Objekt trifft. Die Berechnung des Reflexionsvektors erfolgt in Weltkoordinaten. In Abbildung 52 ist im Weltkoordinatensystem dargestellt, wie der Vektor **U** durch **PosKam-PosW** berechnet werden kann.

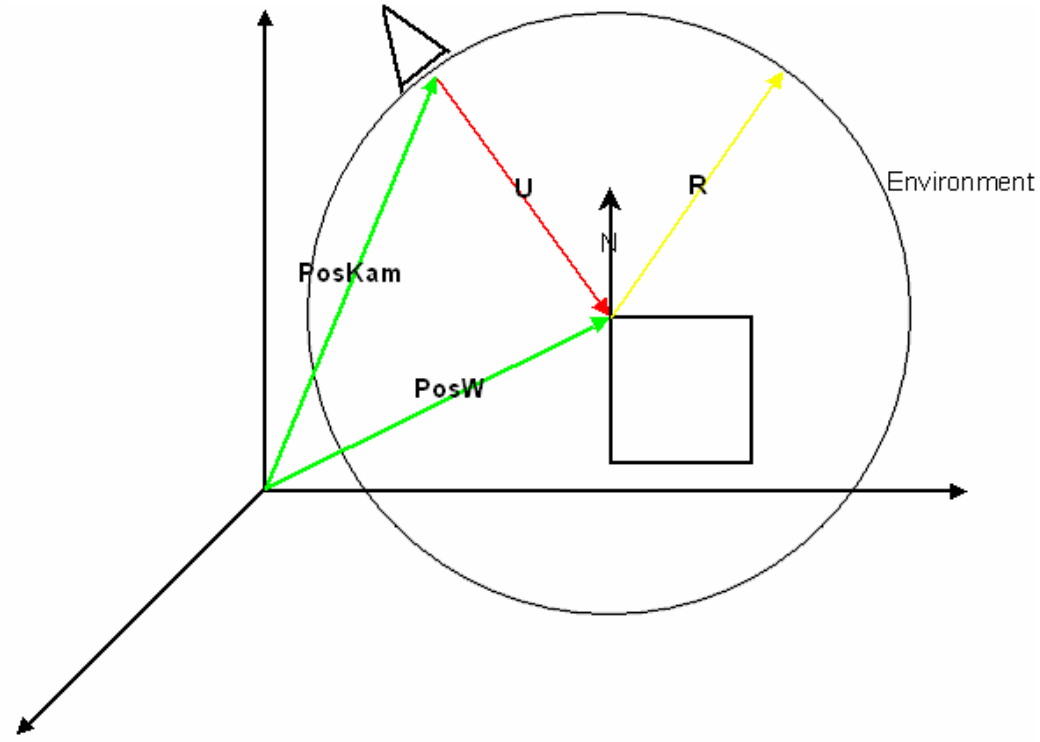


Abbildung 52: Berechnung des Reflexionsvektors

PosW ist die Position des Fragments in Weltkoordinaten.

```
//Position des Pixels in Weltkoordinaten
float4 PosW = mul(world, PosO);
```

Listing 6: Umrechnung der Position in Weltkoordinaten

PosKam ist die Position der Kamera in Weltkoordinaten. Sie findet sich in der Transformationsmatrix wieder, die vom Kamerakoordinatensystem in das Weltkoordinatensystem umrechnet. Denn wie aus Formel 11 ersichtlich, beinhaltet die letzte Spalte der Transformationsmatrix den Vektor, um den ein Koordinatensystem verschoben wird. Da die Translation eines Koordinatensystems um einen beliebigen Vektor **v** äquivalent ist zu einer Objekttranslation um den Vektor **–v**, muss hier die inverse Matrix der view-Matrix gewählt werden [26]. Der Vektor **U** berechnet sich also durch:

```
float3 U = normalize(float3(viewinverse[0].w,
                            viewinverse[1].w,
                            viewinverse[2].w)-PosW.xyz);
```

Listing 7: Berechnung des Vektors U

Um den Reflexionsvektor von **U** über den Normalenvektor **N** zu berechnen, gibt es in Cg die Methode *reflect()*:

```
float3 R = reflect(U, Normal);
```

Listing 8: Berechnung des Vektors R

R wird in der gleichen Länge wie **U** ausgegeben. Da **R** normiert vorliegen muss, wurde **U** zuvor normiert.

Nachdem der Reflexionsvektor berechnet wurde, muss bestimmt werden, auf welche Texturkoordinate der Umgebung er zeigt.

Vereinfacht kann man sich vorstellen, dass das Fragment, welches gerade berechnet wird, immer im Mittelpunkt der Umgebungs-Kugel (vgl. auch Abbildung 52) liegt, da eine orthografische Projektion [7] vorliegt. Verglichen mit Abbildung 7 kann **R** also in Kugelkoordinaten dargestellt und die Formel 7 und Formel 8 genutzt werden. Die

[7] Bei einer orthografischen Projektion ist der Ansichtsbereich wie ein Kasten geformt. Anders als bei der perspektivischen Projektion ändert sich so die Größe des Ansichtsbereiches von einem Ende zum Anderen nicht. Folglich beeinflusst der Abstand von der Kamera nicht, wie groß ein Objekt erscheint. [27]

Umrechnung des Punktes auf der Kugel in die Texturkoordinaten wird anhand der Formel 9 und 10 realisiert.

```
float l = atan2(R.y,R.x);                  //Längengrad
float b = acos(-R.z);                      //Breitengrad

float u = ((l+pi)/(2*pi));
float v = (b/pi);                          //u und v im Intervall [0,1]
float2 texcoord = float2(u,v);
```

Listing 9: Berechnung der Texturkoordinaten auf der Umgebungstextur

Bei der Koordinate u ist noch zu beachten, dass die Methode atan2 (in Cg vorhanden) ein Intervall [-π, π] zurück gibt, für u aber das Intervall [0, 1] benötigt wird. Deshalb muss auf den Längengrad π addiert werden, bevor er durch 2 π geteilt wird.
Um die Umgebungstextur in ihrer Lage, ihres Drehwinkels oder der Skalierung zu beeinflussen, werden anschließend drei Matrizen erstellt, wozu die Methoden „scalematrix()“, „transmatrix()“ und „rotmatrix()“ genutzt werden. Die Texturkoordinaten werden mit diesen Matrizen multipliziert, so kann die Umgebungstextur über das User Interface beeinflusst werden. Die genutzten Matrizen findet man in Formel 11, Formel 12 und Formel 13.
In der Methode „multitex()“ werden die Texturen mit einer Addition zusammen gemischt.

```
float4 multitex(float4 col1, float4 col2){
      float4 color = col2;
      if(TextureMix==true)
                  {color=(col2 * mixfactor + col1 * (1.0f - mixfactor));}
      return color;
}
```

Listing 10: Mischung der Texturen

Der *mixfactor* bestimmt damit noch einmal die Intensität der Texturen. Beide Intensitäten ergeben zusammen immer Eins, also 100 Prozent. So wird bei einem *mixfactor* größer Eins die abgebildete Textur von der Umgebungstextur abgezogen, kleiner Null wird die Umgebungstextur abgezogen.
Als letzter Abschnitt wird in der CgFX-Datei die Technik erstellt. Hier wurde das ARB-Profil genutzt, damit dies mit den Gegebenheiten des Cg- Shader im 3Designer übereinstimmt.

Der komplette Code des Shaders ist mit dem Dateinamen „optim_sphere_environment.cgfx“ auf der beiliegenden CD zu finden. Eine kurze Anleitung zur Nutzung des Shaders befindet sich ebenfalls im Anhang.

7.2 Cg für 3Designer

Der Cg-Shader ist ähnlich aufgebaut, wie der CgFX-Shader. Es gibt allerdings systembedingt Unterschiede.
Da es sich um Cg-Shader handelt, müssen zwei Files benutzt werden, ein Vertex- und ein Fragment-Shader. Im Vertex-Shader wird eine Struktur für den Input und eine für den Output erstellt. Sowohl der Output als auch der Input haben mehr Parameter als der CgFX-Shader. Dem liegt zugrunde, dass in diesem Shader sowohl Farben, als auch Multitexturen und Lichter mit einberechnet werden können.
Zusätzlich zu den Parametern im CgFX-Shader werden dafür *Color*, *Texcoord1* und *Texcoord2* in den Shader eingegeben.

```
struct VS_INPUT
  {
      float4 Position   : POSITION0;
      float3 Normal     : NORMAL;
      float4 Color      : COLOR0;
      float4 Texcoord0  : TEXCOORD0;
      float4 Texcoord1  : TEXCOORD1;
      float4 Texcoord2  : TEXCOORD2;
  };

struct VS_OUTPUT
  {
      float4 Color            : COLOR0;
      float4 Position         : POSITION0;
      float4 Texcoord0        : TEXCOORD0;
      float4 Texcoord1        : TEXCOORD1;
      float4 Texcoord2        : TEXCOORD2;
      float4 PosO             : TEXCOORD3;
      float4 VertexPosition   : TEXCOORD4;
      float3 Normal           : TEXCOORD5;
      float3 Tangent          : TEXCOORD6;
      float3 Binormal         : TEXCOORD7;
  };
```

Listing 11: Vertex-Input und Vertex-Output

Im Output sind diese drei Parameter ebenfalls vorhanden, zusätzlich die Parameter *Tangent*, *VertexPosition* und *Binormal*, die zur Lichtberechnung benötigt werden.

Der Vertex-Shader transformiert den Normalenvektor des Vertex in Kamerakoordinaten. Über *Position* wird die Position des Vertex in Projektionskoordinaten umgerechnet. Die beiden Matrizen stehen über OpenGL zur Verfügung (vgl. Abbildung 32). *PosO* gibt, wie im CgFX-Shader, die Position in Objektkoordinaten weiter.
Die Farbe des Vertex *colo*r wird ohne Veränderung weiter gegeben.
Die Texturkoordinaten werden in einem OpenGL-State gespeichert. Änderungen, die über das Texturbedienfeld an den Texturen gemacht werden, werden hier ebenfalls gespeichert. Der Shader greift auf den OpenGL-State zu und übernimmt damit die Textur in dem Zustand, wie sie im Bedienfeld eingestellt wurde.

```
VS_OUTPUT main( VS_INPUT Input)
  {
      VS_OUTPUT Output;

      //Normalenvektor in Kamerakoordinaten
      float3 Normal =
      normalize(mul((float3x3)glstate.matrix.invtrans.modelview[0],
      Input.Normal));
      //Position in Projektionskoordinaten
      Output.Position = mul(glstate.matrix.mvp, Input.Position );
      Output.Normal = Normal;
      //Position in Objektkoordinaten
      Output.PosO   = Input.Position;

      float3 Tangent  = -float3(abs(Normal.y) + abs(Normal.z),
                        abs(Normal.x), 0);
      float3 Binormal = -float3(0, abs(Normal.z), abs(Normal.x) +
                        abs(Normal.y));
      float4 EyePos   = mul(glstate.matrix.modelview[0], Input.Position);

      Output.Texcoord0= mul( glstate.matrix.texture[0], Input.Texcoord0);
      Output.Texcoord1= mul( glstate.matrix.texture[1], Input.Texcoord1);
      Output.Texcoord2= mul( glstate.matrix.texture[2], Input.Texcoord2);
      Output.Tangent  = Tangent;
      Output.Binormal = Binormal;
      Output.VertexPosition = EyePos;
      Output.Color    = Input.Color;

      return Output;
  }
```

Listing 12: Vertex-Shader

Im Fragment-Shader werden diese Texturen über die Semantic TEXUNIT0 bis TEXUNIT3 zugänglich gemacht. Der Rest des Inputs für den Fragment-Shader wird vom Output des Vertex-Shaders übernommen.
Nach dem Input werden die Cg-Files commonFragment.cg und calc_lights.cg durch den Befehl „#include“ für diesen Shader zugänglich gemacht. CommonFragment.cg dient der

Nutzung von Multitexturen und verrechnet diese miteinander. In calc_lights.cg werden die in der Szene definierten Lichter in den Shader einberechnet.
Die Methode „float4 main()“ ist der eigentliche Shader. Hier werden neben dem Input die uniform-Parameter in den Shader eingegeben.

```
float4 main( PS_INPUT Input,
                  uniform sampler2D environmentTexture,
                  uniform float transformU,
                  uniform float transformV,
                  uniform float scaleU,
                  uniform float scaleV,
                  uniform float rotate,
                  uniform float intensity,
                  uniform float mixfactor,
                  uniform float pan
                  ) : COLOR0
```

Listing 13: uniform-Parameter

Die uniform-Parameter sind die Verbindungen zum User Interface (vgl. Kapitel 4.2) und erstellen dieses, wie in Abbildung 53 dargestellt.

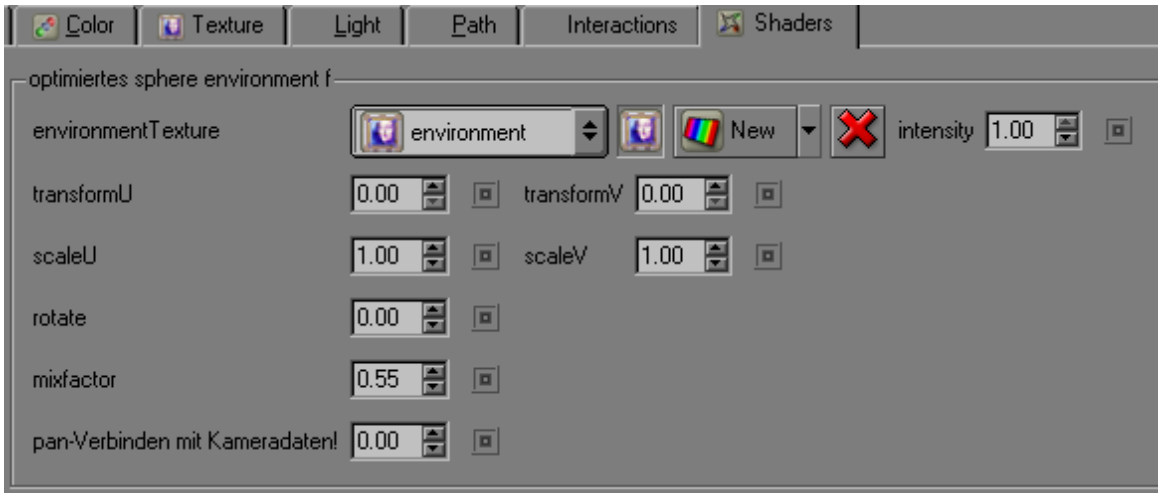


Abbildung 53: User Interface 3Designer

Die Methode „calc_sphere_coords“ berechnet, ebenso wie in CgFX, die Texturkoordinaten auf der Umgebungstextur, auf die der Reflexionsvektor zeigt. Auch wird mit der Methode „tex2D()“ die Farbe zu den jeweiligen Texturkoordinaten berechnet Die Intensität der Umgebungstextur wird hinzugerechnet und in *sphereRGBA* gespeichert. Sobald das Objekt mit einer Textur belegt ist, kann diese mit dem *mixfactor* mit der Umgebungstextur genau wie im CgFX-Shader verrechnet werden.

Die Berechnung der Koordinate auf der Umgebungstextur in der Methode „calc_sphere_coords“ läuft allerdings anders ab, als in dem CgFX-Shader. Hier mussten Eingeständnisse gemacht werden, da die Berechnung von Vektoren nicht in Weltkoordinaten, sondern in Kamerakoordinaten gerechnet werden muss. Da die Kamera in Kamerakoordinaten im Ursprung des Koordinatensystems liegt, ist die Position des Fragments in Kamerakoordinaten der Vektor, der von der Kamera auf das Fragment zeigt. **U** berechnet sich in diesem Shader also aus der Position des Fragments in Kamerakoordinaten mit Hilfe der Multiplikation der Matrix glstate.matrix.modelview[0] auf die Position in Objektkoordinatensystem.

```
//Position in Kamerakoordinaten
float3 U = normalize(mul(glstate.matrix.modelview[0],PosO).xyz);
```

Listing 14: Berechnung des Vektors U

Der Unterschied in der Berechnung von **U** in diesem Shader wirkt sich auf den Reflexionsvektor **R** aus. Er wird zwar wie in CgFX implementiert (wobei die Änderung der Koordinatenachsen mit einbezogen werden muss), jedoch wird er durch das fehlende Weltkoordinatensystem immer relativ zur Kamera berechnet und die Kamerabewegung kann nicht mit in diese Berechnung eingehen. Die Kamera sieht immer den gleichen Teil der Umgebungstextur.
Um die Kamerabewegung dennoch in **R** mit einzuberechnen wurde folgender Trick angewandt:
Über die Connections, die schon in Kapitel 3.2 beschrieben wurden, können die Kameradaten mit dem Shader verbunden werden.
Als Output-Connection dient die Pan-Bewegung (also die Drehung der Kamera in horizontaler Richtung), sie wird über eine Verbindung auf dem User Interface des Shaders dem Shader als Input gegeben und in diesem Teil des Codes verrechnet:

```
//Rotationsmatrix für R verbunden mit pan der Kamera (um y-Achse
rotieren)
      pan = pan/-50;
      float4x4 rotationy = {cos(pan), 0, sin(pan), 0,
                            0, 1, 0, 0,
                            -sin(pan), 0, cos(pan), 0,
                            0, 0, 0, 1};
      R = normalize(mul((float3x3)rotationy,R));
```

Listing 15: Einberechnung der Rotation der Kamera

Die Rotation der Kamera um die y-Achse wird in eine Rotationsmatrix geschrieben und auf den Vektor **R** gerechnet. Zusätzlich zur Pan-Bewegung könnte die Tilt-Bewegung der Kamera, als Rotation von **R** um die x-Achse, mit einberechnet werden. Da im Studio diese Bewegung allerdings eingeschränkt eingesetzt wird und so die fehlende Rotation im Auge des Betrachters nicht irritiert, wurde sie nicht mit einberechnet.
Weitere Überlegungen, wie mit den technischen Einschränkungen umgegangen werden kann, finden sich in Kapitel 8 Zusammenfassung und Ausblick.
Das Ende dieser Methode gleicht wieder der Methode im CgFX-Shader. Es werden drei Matrizen erstellt, die zum Skalieren, Rotieren und Verschieben der Umgebungstextur dienen und über das User Interface des Shaders beeinflusst werden.

Um, wie gefordert, Multitexturen, gesetzte Lichter in einer Szene oder Materialeigenschaften eines Objektes bei der Nutzung des Shaders mit einzuberechnen, werden in diesem Shader OpenGL-States und Präprozessor-Statements verwendet.
Präprozessoren lesen die Präprozessor-Statements und verarbeiten sie. Sobald ein Parameter gesetzt oder in seinem Zustand geändert wurde, soll dieser mit in den Shader eingerechnet werden. Hier ein Beispiel, wie dies mit Textur-States implementiert wird:

```
#ifdef RE_TEXUNIT1_2D
      //Farbe auf der zweiten Textur
      float4 colTex1 = tex2D(Input.BaseMap1, Input.Texcoord1 );
      //verrechnet die beiden Farben der Texturen
      texturRGBA = mtex1(colTex1, texturRGBA);
#endif
```

Listing 16: Präprozessor für Textur

Mit #ifdef wird abgeprüft, ob zusätzliche Texturen für das Objekt definiert wurden und berücksichtigt werden müssen. Wenn dies der Fall ist, so soll die Farbe in dem Fragment der Textur (colTex1) mit der Methode „mtex1()“ mit den anderen Texturen verrechnet werden. Diese Methode findet sich in der Datei commonFragment.cg wieder, die für die Multitexturierung zuständig ist. Mit #endif wird der bedingte Codeabschnitt abgeschlossen. Wurde keine Textur definiert, so geht der Code zwischen #ifdef RE_TEXUNIT1_2D und #endif nicht in den Shader mit ein.
Eine Liste der Präprozessor-Statements, die in einem Shader für 3Designer genutzt werden können, befindet sich im Anhang 9.3. In diesem Shader werden Präprozessor-Statements für Lichter, Materialeigenschaften und Texturen verwendet. Es können drei Texturen für

eine Multitexturierung und jeweils acht verschiedene Lichtquellen als Punktlicht, Spotlicht und Grundlicht verwendet werden.

Der Code für die Shader und die zusätzlichen Cg-Files commonFragment.cg und calc_lights.cg sind auf der beiliegenden CD zu finden.
Eine kurze Bedienungsanleitung für diesen Shader ist im Anhang verzeichnet.

8 Fazit und Aussichten

Das Ziel der Arbeit bestand darin zwei Environment-Shader zu entwickeln, die Reflexionen im Raum auf reflektierenden Oberflächen sichtbar machen. Bei der Umsetzung sollte der Workflow zur Erstellung virtueller Szenenbilder im WDR Köln berücksichtigt werden.

Dieses Ziel wurde erreicht. Für die Programme Maya und 3Designer liegt jeweils ein Shader vor. In den Shadern wird um ein Objekt eine Kugel gelegt, welche die Umgebung darstellt, die reflektiert werden soll. Die dafür zu nutzende Textur kann mit Hilfe einer Chromkugel erstellt werden, die in das virtuelle Szenenbild gelegt wird und die gesamte Umgebung spiegelt.

Durch die Nutzung einer Kugel als Umgebung ist die Unabhängigkeit der Blickrichtung gewährleistet. Aus unterschiedlichen Richtungen sieht man auch unterschiedliche Ausschnitte der Umgebungstextur. Um den reflektierten Ausschnitt weiter zu beeinflussen, kann die Textur über das User Interface gedreht, skaliert oder verschoben werden. Auch kann die Reflexion mit einer auf dem Objekt bereits vorhandenen Textur gemischt werden.

Da in der Implementierung des Shaders für 3Designer keine Weltkoordinaten zur Verfügung stehen, muss die Kamerarotation über Connections auf dem User Interface in den Shader einbezogen werden.

Eine Möglichkeit, ohne die Connections im User Interface des 3Designer auszukommen, könnte darin bestehen, auf einem anderen Weg an die Kamerabewegungen heran zu kommen und diese direkt in den Shader mit einzubeziehen. Sollte ein Weg zu finden sein, die Kameraposition in Weltkoordinaten und ihre Rotation zum Weltkoordinatensystem heraus zu finden, kann im Shader eine Transformationsmatrix geschrieben werden, die in Weltkoordinaten umrechnet. So könnte die Berechnung dieses Shaders ebenfalls in Weltkoordinaten umsetzbar sein.

In der Abteilung zur Erstellung der Szenenbilder und der Studioproduktion virtueller Sendungen des WDR Köln kann nach dieser Arbeit mit den Shadern gearbeitet werden. Zuvor waren die Szenenbilder durch fehlende Reflexionen während einer Kamerabewegung durch das Szenenbild, starr. Es gab feste Texturen auf den Objekten, auf denen Schatten und Lichter vorhanden waren, doch keine bewegten Reflexionen. Nun erscheinen die Szenenbilder bei Kamerafahrten „lebendiger“.

9 Anhang

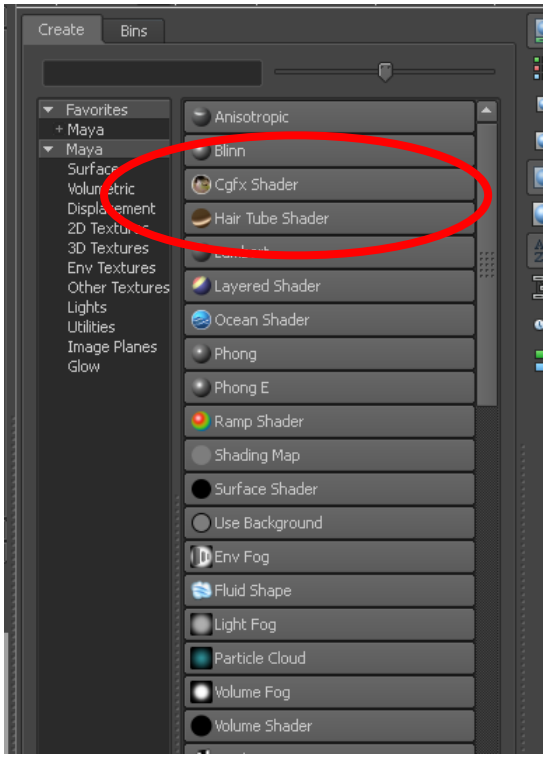
9.1 Anleitung zur Nutzung des CgFX-Shaders

Anleitung zur Nutzung des CgFX-Shaders „Spherical Environment Mapping“ in Maya:

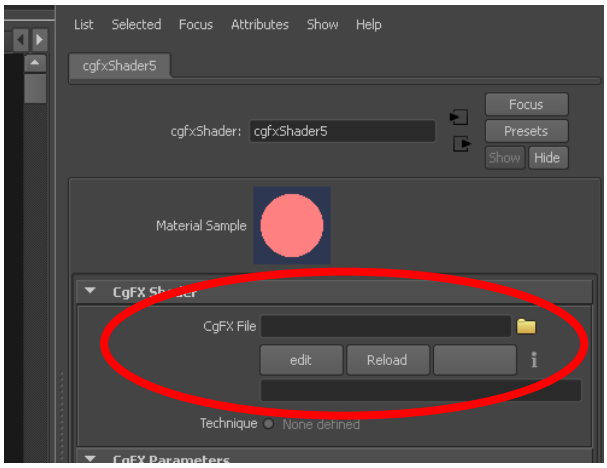
1.) Einbinden des Shaders

Um den Shader einzubinden muss das Plugin für die Nutzung von CgFX-Shadern unter ***„Window“ → „Settings/Preferences“ → „Plug-in Manager“*** aktiviert sein.

Sobald das PlugIn aktiviert ist, kann im Hypershade unter den Render Nodes ein neuer CgFX Shader erstellt werden.



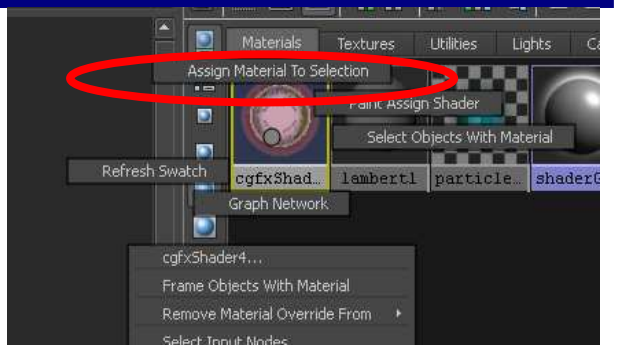
Im Attribute Editor muss der Pfad zu der CgFX-Datei angegeben werden.



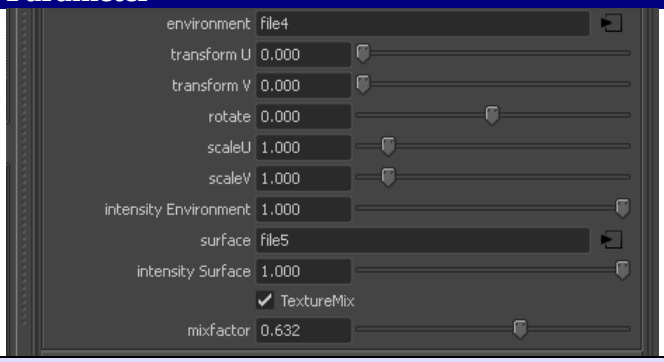
Sobald der Shader eingebunden ist, erscheint im Attribute Editor das User Interface.

2.) Anwendung auf ein Objekt

Um den Shader auf ein Objekt anzuwenden muss dieses markiert sein. Rechtsklick mit der Maus auf den Shader im Hypershade und die Taste gedrückt halten, öffnet ein Menü, wo der Punkt „Assign Material to Selection“ angewählt werden muss.



3.) Einstellbare Parameter



User Interface	Beschreibung
Environment Textur	Einbinden der Umgebungstextur
transform U	Verschiebung der Umgebungstextur in horizontaler Richtung
transform V	Verschiebung der Umgebungstextur in vertikaler Richtung
rotate	Drehung der Umgebungstextur
scale U	Skalierung der Umgebungstextur in horizontaler Richtung 1 → Originalgröße
scale V	Skalierung der Umgebungstextur in vertikaler Richtung 1 → Originalgröße
Intensity Environment	Intensität der Reflexion: 0 → keine Reflexion 1 → 100% Reflexion
surface	Einbindung der abgebildeten Textur
Intensity surface	Intensität der Textur auf der Oberfläche: 0 → keine Textur sichtbar 1 → 100% der Textur sichtbar
TextureMix	Wird dieser Hacken gesetzt, besteht die Möglichkeit beide Texturen miteinander zu mischen.
mixfactor	Mischt die Texturen miteinander: 0 → nur die abgebildete Textur ist zu sehen, keine Reflexionen 1 → nur die Reflexionen ist zu sehen kleiner 0 → Subtraktion der Reflexion von der abgebildeten Textur größer 1 → Subtraktion der abgebildeten Textur von der Reflexion

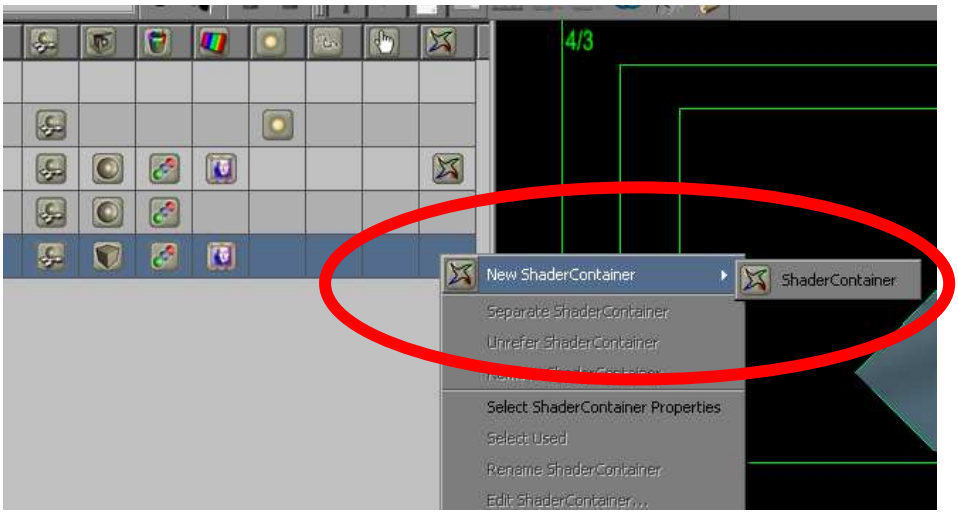
9.2 Anleitung zur Nutzung des Cg-Shaders

Anleitung zur Nutzung des Cg-Shaders „Spherical-Environment-Mapping“ im 3Designer:

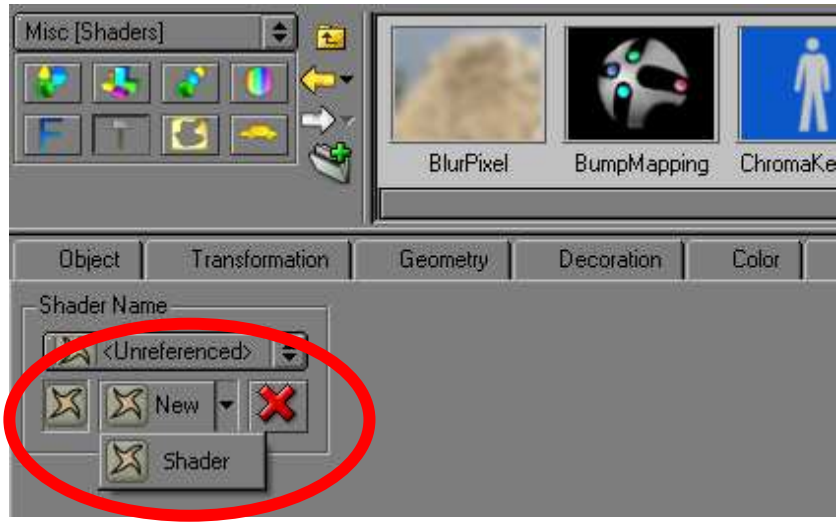
1.) Einbinden des Shaders

Um den Shader einzubinden müssen die Cg-Dateien: *optimiertes_sphere_environment_f.cg*, *optimiertes_sphere_environment _v.cg*, *calc_lights.cg* und *commonFragment.cg* unter *G:\Shaders* gespeichert sein.

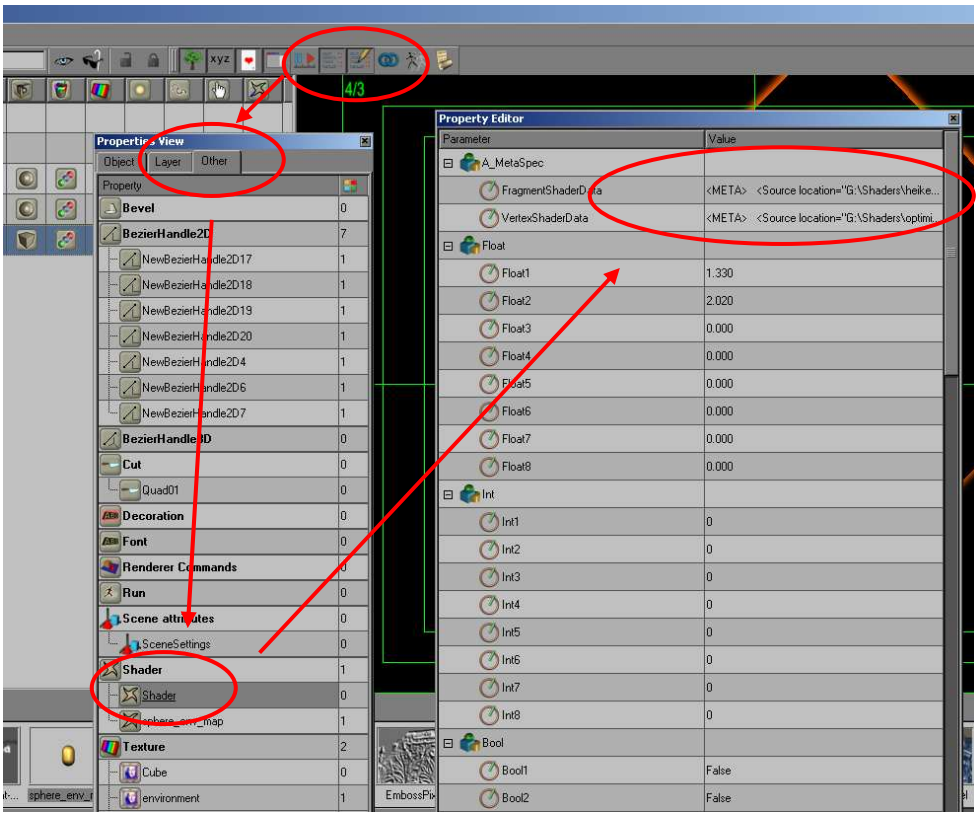
Rechtsklick auf den ShaderContainer-Reiter des Objektes und einen neuen ShaderContainer erstellen.



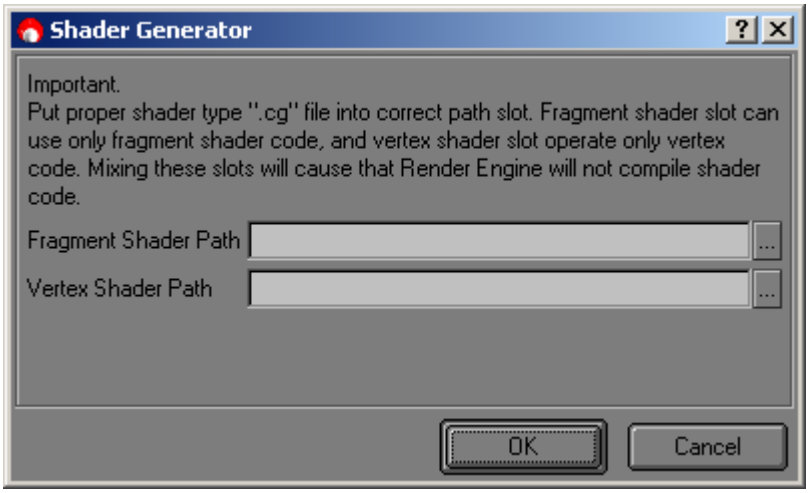
Über „New“ den neuen Shader einbinden.



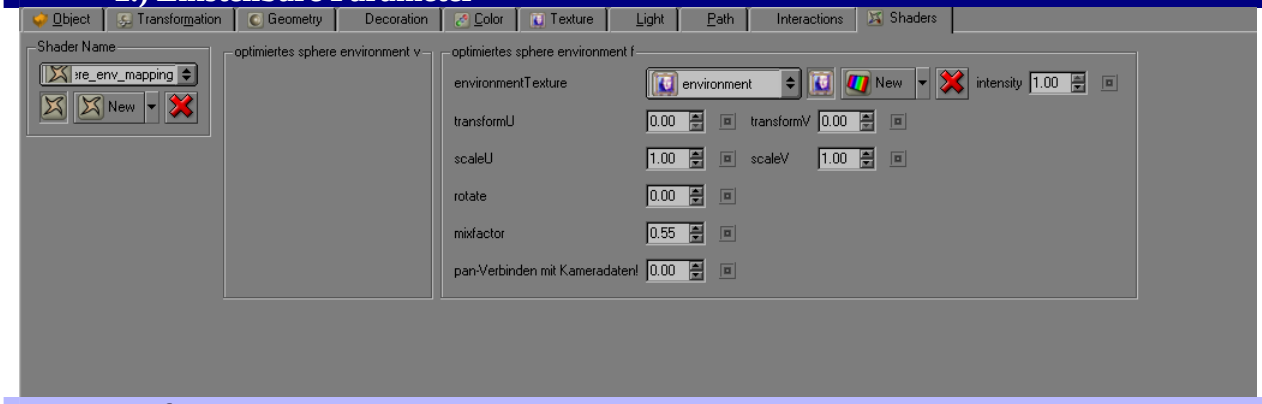
Im Advanced Editor im PropertiesView unter „Other“ den Shader markieren und im Property Editor in die Vertex- und Fragmentshader-datenfeldern die XML-Anweisungen eingeben. Dies stellt die Verbindung zur Cg-Datei her.



Sollte der zuvor aufgezeigte Weg nicht funktionieren über den Menüpunkt „New“ unter „Shader“ den Fragment- und Vertex-Shader einbinden. Anschließend die XML-Datei wie oben beschrieben einfügen.



2.) Einstellbare Parameter

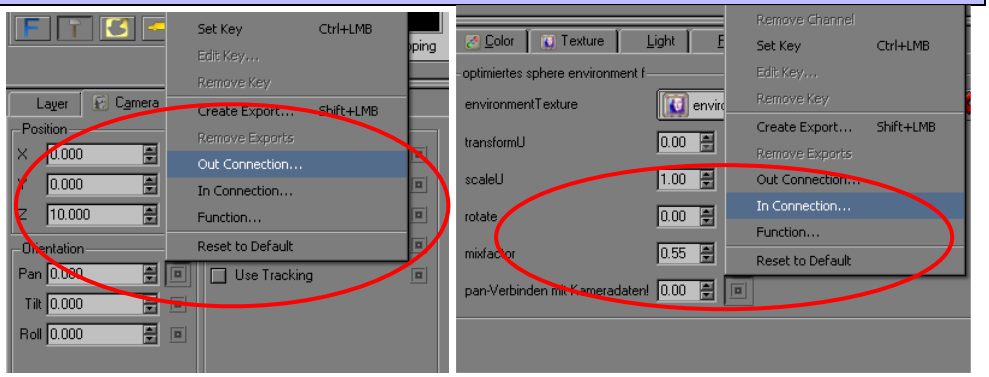


User Interface	Beschreibung
Environment Textur	Einbinden der Umgebungstextur
transform U	Verschiebung der Umgebungstextur in horizontaler Richtung
transform V	Verschiebung der Umgebungstextur in vertikaler Richtung
rotate	Drehung der Umgebungstextur
scale U	Skalierung der Umgebungstextur in horizontaler Richtung 1 → Originalgröße
scale V	Skalierung der Umgebungstextur in vertikaler Richtung 1 → Originalgröße
intensity	Intensität der Reflexion: 0 → keine Reflexion 1 → 100% Reflexion
mixfactor	Mischt die Texturen miteinander: 0 → nur die abgebildete Textur ist zu sehen, keine Reflexionen 1 → nur die Reflexionen ist zu sehen kleiner 0 → Subtraktion der Reflexion von der abgebildeten Textur größer 1 → Subtraktion der abgebildeten Textur von der Reflexion
pan-verbinden mit Kameradaten!	Dieser Regler dient dazu die horizontale Kamerabewegung in den Shader mit einzuberechnen und sollte nicht benutzt werden. Vor der Nutzung des Shaders muss eine Connection mit dem Output von der Pan- Bewegung der Kamera zu diesem Parameter gemacht werden.

Erstellung der Connection:

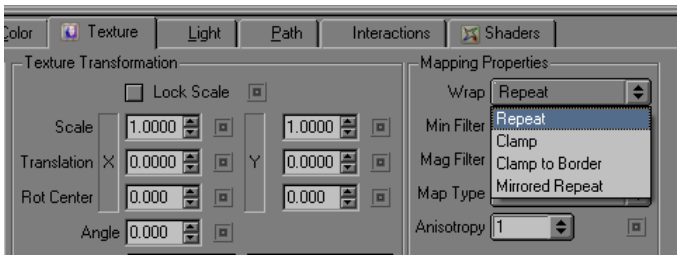
Output:
Kameradaten
„Pan“

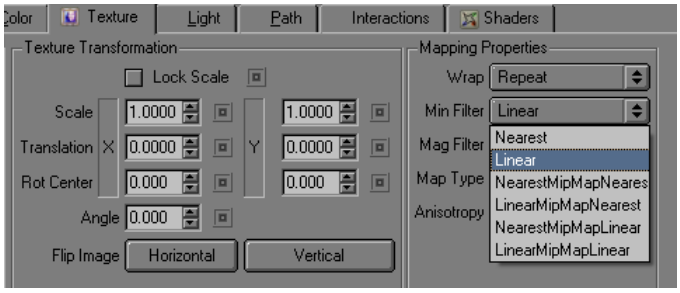
Input:
Shaderdaten
„pan-Verbindung“

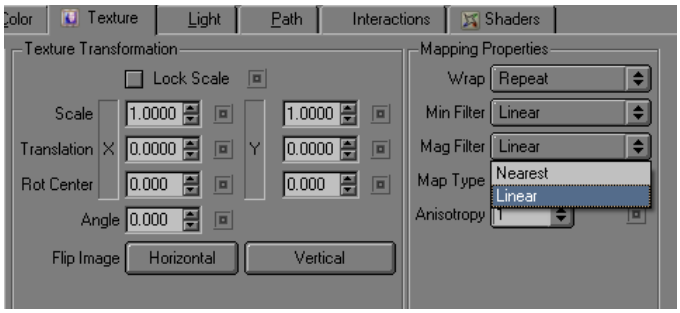


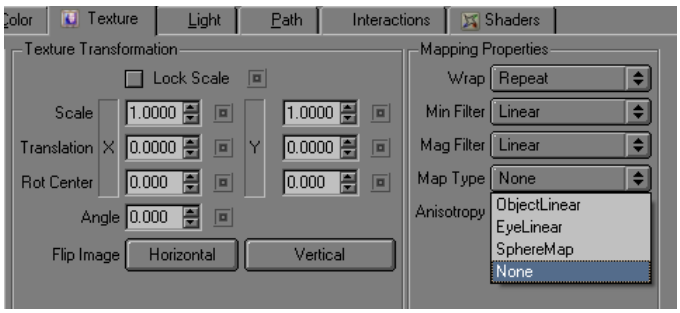
Um die Gleichheit der Darstellung dieses Shaders mit dem CgFX-Shader zu gewährleisten müssen die Texturfilter an die des CgFX-Shaders angepasst werden zu:

Wrap: Repeat
Min Filter: Linear
Mag Filter: Linear
Map Type: None









9.3 Liste der Präprozessor-Statements (im 3Designer nutzbar) [24]

Preprocessor directive	Description
RE_PLAIN_COLOR	Plain color is enabled
RE_MATERIAL	Material is enabled
RE_LIGHT0_INFINITE ... RE_LIGHT7_INFINITE	Light(s) [0..7] as Infinite type is(are) enabled
RE_LIGHT0_POINT ... RE_LIGHT7_POINT	Light(s) [0..7] as Point type is(are) enabled
RE_LIGHT0_SPOT ... RE_LIGHT7_SPOT	Light(s) [0..7] as Spot type is(are) enabled
RE_TEXUNIT0_2D ... RE_TEXUNIT3_2D	[0..3] texture unit(s) is(are) enabled with 2D textures
RE_TEXUNIT0_3D ... RE_TEXUNIT3_3D	[0..3] texture unit(s) is(are) enabled with 3D textures
RE_TEXUNIT0_CUBE ... RE_TEXUNIT3_CUBE	[0..3] texture unit(s) is(are) enabled with CUBE textures
RE_TEXCOORD0_EYE ... RE_TEXCOORD3_EYE	[0..3] texture EYE coordinates are enabled
RE_TEXCOORD0_OBJECT ... RE_TEXCOORD3_OBJECT	[0..3] texture OBJECT coordinates are enabled
RE_TEXCOORD0_SPHERE ... RE_TEXCOORD3_SPHERE	[0..3] texture SPHERE coordinates are enabled
RE_TEXCOORD0_NORMAL ... RE_TEXCOORD3_NORMAL	[0..3] texture NORMAL coordinates are enabled
RE_TEXCOORD0_REFLECTION ... RE_TEXCOORD3_REFLECTION	[0..3] texture REFLECTION coordinates are enabled
RE_MTEX0_FUNC_ADD ... RE_MTEX3_FUNC_ADD	[0..3] texture unit(s) ADD function is selected
RE_MTEX0_FUNC_SUB ... RE_MTEX3_FUNC_SUB	[0..3] texture unit(s) SUB function is selected
RE_MTEX0_FUNC_MUL ... RE_MTEX3_FUNC_MUL	[0..3] texture unit(s) MUL function is selected

RE_MTEX0_FUNC_MIX ... RE_MTEX3_FUNC_MIX	[0..3] texture unit(s) MIX function is selected. An additional uniform float parameter is required for this function (IS NOT SUPPORTES YET)
RE_MTEX0_FUNC_OVER ... RE_MTEX3_FUNC_OVER	[0..3] texture unit(s) OVER function selected
RE_MTEX0_FUNC_OFF ... RE_MTEX3_FUNC_OFF	[0..3] texture unit(s) OFF function is selected
RE_OBJ_TEXT_TEXTURED	A text object with a texture decoration assigned is currently rendered.
RE_OBJ_TELESTRATOR	A telestrator object is currently rendered. The object is specific because it enables 3 first texture units with 2D textures assigned to them.

9.4 Literaturverzeichnis:

[1] Prof. G. Zachmann, „Computergrafik I“, „Computergrafik II“ WS 2010/11 Clausthal Universität, Institut für Informatik

[2] Doz. Dr. P. Schenzel, „Einblicke in die GPU-Programmierung“ WS 2006, Universität Halle-Wittenberg, Institut für Informatik

[3] Bender, M., Brill, M.: „Computergrafik – ein anwenderorientiertes Lehrbuch“. Carl Hanser Verlag München Wien, 2006

[4] Prof. S. Grünvogel, „Computeranimation“ SS 2010 FH Köln, Institut für Medien- und Phototechnik

[5] Nischwitz, A., Haberäcker, P.: „Masterbuch Computergrafik und Bildverarbeitung“. Vieweg-Verlag, 2004

[6] Vornberger, O., Müller, O.: „Scan line Verfahren für Polygone“. Stand 2000 URL:http://www-lehre.informatik.uni-osnabrueck.de/~cg/2000/skript/4_2_Scan_Line_Verfahren_f_252_r.html (letzter Aufruf am 20.02.2011)

[7] Salter, T.: „Beauty is only pixel deep“. Stand 2011 URL: http://www.ntsc-uk.com/features/tec/BeautyPixelDeep/GouraudPhong.jpg (letzter Aufruf am 20.02.2011)

[8] Dr. S. Krömker, „Computergrafik I“ WS 2007/08 Universität Heidelberg, Institut für Wissenschaftliches Rechnen

[9] Albinger, R., Alvir, O., Reithofer, J.: „Kugelkoordinaten und die Erde“. Stand 2010 URL: https://elearning.mat.univie.ac.at/physikwiki/index.php/LV009:LV-Uebersicht/WS09_10/Arbeitsbereiche/Kugelkoordinaten_und_die_Erde (letzter Aufruf am 21.02.2011)

[10] Prof. S. Grünvogel, „Vorlesung CGI“ WS 2007/08 FH Köln, Institut für Medien- und Phototechnik

[11] ati.com: “visuelle Details – aus jedem Pixel das Letzte herausholen”. URL: http://www.3dchip.com/Technologie/ATI%20CHIP%20TECHNOLOGY.php (letzter Aufruf am 21.02.2011)

[12] Hardware - das Hardwaremagazin für PC-Spieler. URL: http://www.pcgameshardware.de/aid,645483/PCGH-Retro-3D-Lexikon-Teil-4/Grafikkarte/Wissen/bildergalerie/ (letzter Aufruf am 21.02.2011)

[13] Möller, T.: “Real time rendering”.
Peters-Verlag, 2005

[14] Dr. P. Grimm, „Grafische Datenverarbeitung“ SS 2005, FH Erfurt,
Institut für angewandte Informatik

[15] NVIDIA Corporation:
“Cg Toolkit - Users Manual a developer´s Guide to Programmable Graphics”.
September 2005
im Umfang des Cg-Toolkits erhalten

[16] NVIDIA: „NVIDIA Developer Zone“.
URL: http://http.developer.nvidia.com/Cg/Cg_language.html
(letzter Aufruf am 20.02.2011)

[17] NVIDIA: „NVIDIA Developer Zone“.
URL: http://http.developer.nvidia.com/Cg/index_states.html
(letzter Aufruf am 20.02.2011)

[18] Bjorke, K.: „Using SAS with CgFX and FX file formats“. 03/2008
URL: www.developer.nvidia.com/object/using_sas.html

[19] Hoffmann, H.: „Maya 1x1“. Stand 11/2004
URL: http://digital-log.hfg-karlsruhe.de/3d/archives/2004/11/hypershade.html
(letzter Aufruf am 21.02.2011)

[20] ORAD: “Mastering VideoGraphics”. Stand 2010
URL: Orad.tv/products/3Designer
(letzter Aufruf am 20.02.2011)

[21] Hagler, J.: „Koordinatenräume in der Rendering-Pipeline“. Stand 2006
URL: http://www.dma.ufg.ac.at/assets/9320/intern/r_pipeline_tasse.jpg
(letzter Aufruf: 20.02.2011)

[22] Orlamünder, W., Mascolus, B.:
„Computergrafik und OpenGL: Eine systematische Einführung“
Carl Hanser Verlag München Wien, 2004

[23] Autodesk: “Maya Help”. Stand 2009
URL: http://download.autodesk.com/us/maya/2010help/index.html?url=Asts_
Work_with_CgFX_shaders.htm, topicNumber=d0e507005
(letzter Aufruf am 20.02.2011)

[24] ORAD HI-TEC SYSTEMS LTD.: „Render Enging – shaders tutorial“. 11/2008

[25] Böhme, P.: „Programmiersprache C/C++“. Stand 1996
URL: http://www.imb-jena.de/~gmueller/kurse/c_c++/c_struct.html
(letzter Aufruf am 20.02.2011)

[26] Kopp, H.: „Graphische Datenverarbeitung: Methoden, Algorithmen“
und deren Implementierung
Carl Hanser Verlag München Wien, 1989

[27] Prof. Dr.-Ing. R. Schmidt, „virtuelle Realität“ WS 98/99, Hochschule Esslingen,
Informationstechnik

9.5 Abbildungsverzeichnis

9.6 Listing- Verzeichnis:

9.7 Eidesstattliche Erklärung

Ich versichere hiermit, die vorgelegte Arbeit in dem gemeldeten Zeitraum ohne fremde Hilfe verfasst und mich keiner anderen als der angegebenen Hilfsmittel und Quellen bedient zu haben

.

Köln, den 24.02.2011

9.8 Sperrvermerk

Die vorgelegte Arbeit unterliegt keinem Sperrvermerk.

9.9 Weitergabeerklärung

Ich erkläre mich hiermit einverstanden, dass das vorliegende Exemplar meiner Abschlussarbeit oder eine Kopie hiervon für wissenschaftliche Zwecke verwendet werden darf.

Köln, den 24.02.2011